Question Papers Solutions

UNIT 1

1 Define Computer Architecture. Illustrate the seven dimensions of an ISA? (June 2013)(Dec 2013)(Jan 2014)(Jan 2015)(June 2015)(Jan 2016)

The computer designer has to ascertain the attributes that are important for a new computer and design the system to maximize the performance while staying within cost, power and availability constraints. The task has few important aspects such as Instruction Set design, Functional organization, Logic design and implementation.

Instruction Set Architecture (ISA)

ISA refers to the actual programmer visible Instruction set. The ISA serves as boundary between the software and hardware. Th e seven dimensions of the ISA are:

i)Class of ISA: Nearly all ISAs today ar e classified as General-Purpose-Register architectures. The operands are either Registers or Memory locations.
The two popular versions of this class are:
Register-Memory ISAs : ISA of 80x86, can access memory as part of many instructions.
Load -Store ISA Eg. ISA of MIPS, can access memory only with Load or Store instructions.

ii)**Memory addressing**: Byte addressing scheme is most widely used in all desktop and server computers. Both 80x86 and MIPS use byte addressing. Incase of MIPS the object must be aligned. An access to an object of s b yte at byte address A is aligned if A mod s =0. 80x86 does not require alignment. Accesses are faster if operands are aligned.

iii) **Addressing modes**:Specify the address of a M object apart from register and constant operands.
MIPS Addressing modes:
•Register mode addressing
•Immediate mode addressing
•Displacement mode addressing

80x86 in addition to the above addressing modes supports the additional modes of addressing:

i. Register Indirect
ii. Indexed
iii,Based with Scaled index

iv)Types and sizes of operands:

MIPS and x86 support:
•8 bit (ASCII character), 16 bit(Unicode character)
•32 bit (Integer/word)
•64 bit (long integer/ Double word)

•32 bit (IEEE-754 floating point)

•64 bit (Double precision floating point)
•80x86 also supports 80 bit floating point operand.(extended double Precision

v)Operations:The general category o f operations are:

- oData Transfer
- oArithmetic operations
- oLogic operations
- oControl operations
- oMIPS ISA: simple & easy to implement
- ox86 ISA: richer & larger set of operations

vi) Control flow instructions:All ISAs support:
Conditional & Unconditional Branches
Procedure C alls & Returns MIPS 80x86

- Conditional Branches tests content of Register Condition code bits
- Procedure C all JAL CALLF
- Return Address in a R Stack in M

vii) Encoding an ISA

Fixed Length ISA	**Variable Length ISA**
MIPS 32 Bit long	80x86 (1-18 bytes)
Simplifies decoding	Takes less space

Number of Registers and number of Addressing modes hav e significant impact on the length of instruction as the register field and addressing mode field can appear many times in a single instruction.

2.What is dependability? Explain the two measures of Dependability? (June 2014) (Jan 2015)

The Infrastructure providers offer Service Level Agreement (SLA) or Service Level Objectives (SLO) to guarantee that their networking or power services would be dependable.

- Systems alternate between 2 states of service with respect to an SLA:

1. **Service accomplishment**, where the service is delivered as specified in SLA
2. **Service interruption**, where the delivered service is different from the SLA

- Failure = transition from state 1 to state 2
- Restoration = transition from state 2 to state 1

The two main measures of Dependability are Module Reliability and Module Availability. *Module reliability* is a measure of continuous service accomplishment (or time to failure) from a reference initial instant.

1. *Mean Time To Failure* (*MTTF*) measures Reliability
2. *Failures In Time* (*FIT*) = 1/MTTF, the rate of failures

• Traditionally reported as failures per billion hours of operation
• *Mean Time To Repair* (*MTTR*) measures Service Interruption

– *Mean Time Between Failures* (*MTBF*) = MTTF+MTTR
• *Module availability* measures service as alternate between the 2 states of accomplishment and interruption (number between 0 and 1, e.g. 0.9)
• *Module availability = MTTF / (MTTF + MTTR)*

3. Give the following measurements (
Frequency of FP operations=25%
Average CPI of other instructions=1.33
Average CPI of FP operations=4.0
Frequency of FPSQR=2%
CPI of FPSQR=20
Assume that the two design alternative are to decrease the CPI of FPSQR to 2 or to decrease the average CPI of all FP operations to 2.5 compare the two design alternatives using the processor performance equations.
(June 2014) (June 2015) (Jan 2016)

Option 1

$$\text{Speedup}_{FPSQR} = \frac{1}{(1-0.2) + (0.2/15)} = 1.2295$$

Option 2

$$\text{Speedup}_{FP} = \frac{1}{(1-0.5) + (0.5/1.6)} = 1.2307$$

Option 2 is relatively better.

4. Explain in brief measuring, reporting and summarizing performance of Computer. (June 2014) (June 2015)

Performance:

The *Execution time* or *Response time* is defined as the time between the start and completion of an event. The total amount of work done in a given time is defined as the *Throughput*.

The Administrator of a data center may be interested in increasing the *Throughput*. The computer user may be interested in reducing the *Response time*.

Computer user says that computer is faster when a program runs in less time.

$$\text{Performance} = \frac{1}{\text{Execution Time (X)}}$$

The phrase **"X is faster than Y"** is used to mean that the response time or execution time is lower on X than Y for the given task. **"X is n times faster than Y"** means

$$\text{Execution Time}_y = n^* \text{ Execution time}_x$$

$$\text{Performance}_x = n^* \text{ Perfromance}_y$$

The routinely executed programs are the best candidates for evaluating the performance of the new computers. To evaluate new system the user would simply compare the execution time of their workloads.

Benchmarks

The real applications are the best choice of benchmarks to evaluate the performance. However, for many of the cases, the workloads will not be known at the time of evaluation. Hence, the benchmark program which resemble the real applications are chosen. The three types of benchmarks are:

- KERNELS, which are small, key pieces of real applications;
- Toy Programs: which are 100 line programs from beginning programming assignments, such Quicksort etc.,
- Synthetic Benchmarks: Fake programs invented to try to match the profile and behavior of real applications such as Dhrystone.

To make the process of evaluation a fair justice, the following points are to be followed.

- Source code modifications are not allowed.
- Source code modifications are allowed, but are essentially impossible.
- Source code modifications are allowed, as long as the modified version produces the same output.
- To increase predictability, collections of benchmark applications, called *benchmark suites*, are popular
- SPECCPU: popular desktop benchmark suite given by Standard Performance Evaluation committee (SPEC)
  - CPU only, split between integer and floating point programs
  - SPECint2000 has 12 integer, SPECfp2000 has 14 integer programs
  - SPECCPU2006 announced in Spring 2006.

  SPECSFS (NFS file server) and SPECWeb (WebServer) added as server benchmarks

- Transaction Processing Council measures server performance and costperformance for databases
  - TPC-C Complex query for Online Transaction Processing
  - TPC-H models ad hoc decision support
  - TPC-W a transactional web benchmark
  - TPC-App application server and web services benchmark

SPEC Ratio: Normalize execution times to reference computer, yielding a ratio proportional to performance = time on reference computer/time on computer being rated

• If program SPECRatio on Computer A is 1.25 times bigger than Computer B, then

$$1.25 = \frac{SPECRatio_A}{SPECRatio_B} = \frac{\frac{ExecutionTime_{reference}}{ExecutionTime_A}}{\frac{ExecutionTime_{reference}}{ExecutionTime_B}}$$

$$= \frac{ExecutionTime_B}{ExecutionTime_A} = \frac{Performance_A}{Performance_B}$$

Quantitative Principles of Computer Design

While designing the computer, the advantage of the following points can be exploited to enhance the performance.

* **Parallelism:** is one of most important methods for improving performance.

- One of the simplest ways to do this is through pipelining ie, to over lap the instruction Execution to reduce the total time to complete an instruction sequence.
- Parallelism can also be exploited at the level of detailed digital design.
- Set- associative caches use multiple banks of memory that are typically searched n parallel. Carry look ahead which uses parallelism to speed the process of computing.

* **Principle of locality:** program tends to reuse data and instructions they have used recently. The rule of thumb is that program spends 90 % of its execution time in only 10% of the code. With reasonable good accuracy, prediction can be made to find what instruction and data the program will use in the near future based on its accesses in the recent past.

* **Focus on the common case** while making a design trade off, favor the frequent case over the infrequent case. This principle applies when determining how to spend resources, since the impact of the improvement is higher if the occurrence is frequent.

Amdahl's Law: Amdahl's law is used to find the performance gain that can be obtained by improving some portion or a functional unit of a computer Amdahl's law defines the speedup that can be gained by using a particular feature.

Speedup is the ratio of performance for entire task without using the enhancement when possible to the performance for entire task without using the enhancement. Execution time is the reciprocal of performance. Alternatively, speedup is defined as thee

ratio of execution time for entire task without using the enhancement to the execution time for entair task using the enhancement when possible.

Speedup from some enhancement depends an two factors:

i. The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement. Fraction enhanced is always less than or equal to

Example: If 15 seconds of the execution time of a program that takes 50 seconds in total can use an enhancement, the fraction is 15/50 or 0.3

ii. The improvement gained by the enhanced execution mode; ie how much faster the task would run if the enhanced mode were used for the entire program. Speedup enhanced is the time of the original mode over the time of the enhanced mode and is always greater then 1.

$$\text{Execution time}_{new} = \text{Execution time}_{old} \times \left[(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction enhanced}}{\text{Speed up}_{enhanced}} \right]$$

$$\text{Speedup}_{overall} = \frac{\text{Execution time}_{old}}{\text{Execution time}_{new}}$$

The Processor performance Equation:

Processor is connected with a clock running at constant rate. These discrete time events are called clock ticks or clock cycle.

CPU time for a program can be evaluated:

CPU time = CPU clock cycles for a program X clock cycle time

$$\text{CPU time} = \frac{\text{CPU Clock cycles for a program}}{\text{Clock rate}}$$

Using the number of clock cycle and the Instruction count (IC), it is possible to determine the average number of clock cycles per instruction (CPI). The reciprocal of CPI gives Instruction per clock (IPC)

$$\text{CPI} = \frac{\text{CPU clock cycle for a program}}{\text{Instruction count}}$$

CPU time = IC X CPI X Clock cycle time

$$\text{CPU time} = \frac{\text{Seconds}}{\text{program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{clock cycle}}$$

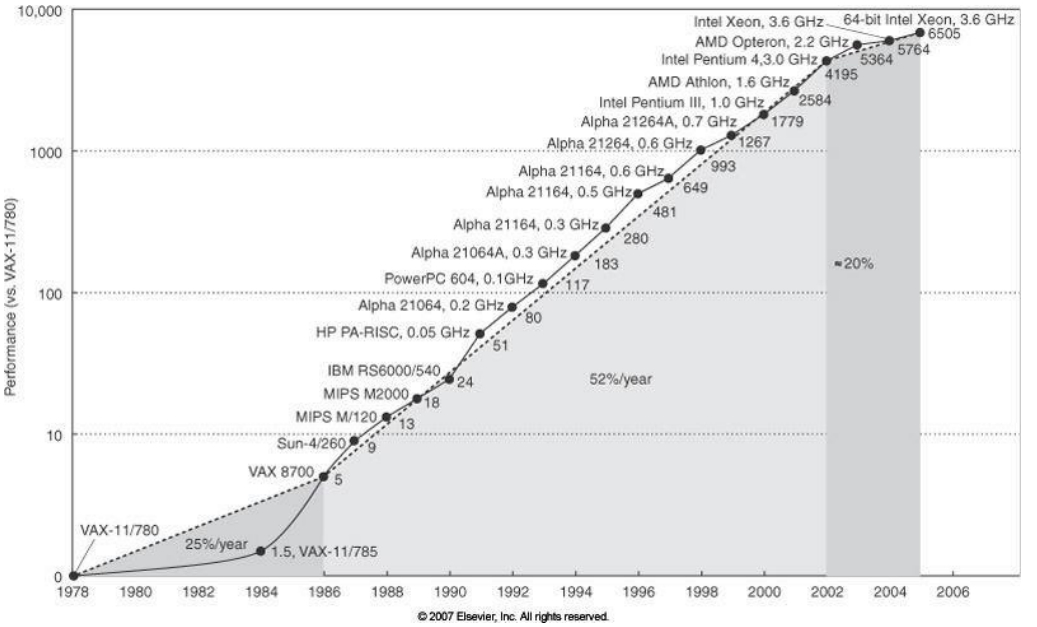
Processor performance depends on IC, CPI and clock rate or clock cycle. There 3 parameters are dependent on the following basic technologies.

Clock Cycle time – H/W technology and organization

CPI- organization and ISA

IC- ISA and compiler – technology

5. Explain with learning curve how the cost of processor varies with time along with factors influencing the cost. (June/July 2013)



1960: Large Main frames (Millions of $)
(Applications: Business Data processing, large Scientific computin g)
1970: Minicomputers (Scientific laboratories, Time sharing concepts)
1980: Desktop Computers (µPs) in the form of Personal computers and workstations.
(Larger Memory, more computing power, Replaced Time sharing g systems)
1990: Emergence of Internet and WWW, PDAs, emergence of high performance digital consumer electronics
2000: Cell phones
These changes in computer use have led to three different computing classes each characterized by different applications, requirements and computing technologies.owth in processor performance since 1980s

6. Find the number of dies per 200 cm wafer of circular shape that is used to cut die that is 1.5 cm side and compare the number of dies produced on the same wafer if die is 1.25 cm. (Jan 2014)(June 2016)

$$\text{Cost of IC} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging and final test}}{\text{Final test yield}}$$

$$\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer x die die yield}}$$

$$\pi \times (\text{wafer diameter}/2)^2$$

$$\text{Dies per wafer} = \text{Die area} - \frac{\pi \times \text{wafer diameter}}{\text{sqrt}(2 \times \text{die area})}$$

7. Define Amdahl's law. Derive n expression for CPU clock as a function of instruction count, clocks per instruction and clock cycle time.
(Jan 2014) (Dec 2013)(Jan 2015)(Jan 2016)

Amdahl's Law: Amdahl's law is used to find the performance gain that can be obtained by improving some portion or a functional unit of a computer Amdahl's law defines the speedup that can be gained by using a particular feature.

Speedup is the ratio of performance for entire task without using the enhancement when possible to the performance for entire task without using the enhancement. Execution time is the reciprocal of performance. Alternatively, speedup is defined as thee ratio of execution time for entire task without using the enhancement to the execution time for entair task using the enhancement when possible.

Speedup from some enhancement depends an two factors:

i. The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement. Fraction enhanced is always less than or equal to

Example: If 15 seconds of the execution time of a program that takes 50 seconds in total can use an enhancement, the fraction is 15/50 or 0.3

ii. The improvement gained by the enhanced execution mode; ie how much faster the task would run if the enhanced mode were used for the entire program. Speedup enhanced is the time of the original mode over the time of the enhanced mode and is always greater then 1.

$$\text{Execution time}_{new} = \text{Execution time}_{old} \times \left[(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction enhanced}}{\text{Speed up}_{enhanced}} \right]$$

$$\text{Speedup}_{overall} = \frac{\text{Execution time}_{old}}{\text{Execution time}_{new}}$$

8. List and explain four important technologies, which has lead to improvements in computer system. (June 2015)(June 2016)

Desktop computing

The first and still the largest market in dollar terms is desktop computing. Desktop computing system cost range from $ 500 (low end) to $ 5000 (high-end configuration). Throughout this range in price, the desktop market tends to drive to optimize price- performance. The performance concerned is compute performance and graphics performance. The combination of performance and price are the driving factors to the customers and the computer designer. Hence, the newest, high performance and cost effective processor often appears first in desktop computers.

Servers:

Servers provide large-scale and reliable computing and file services and are mainly used in the large-scale en terprise computing and web based services. The three important

characteristics of servers are:

•**Dependability**: Severs must operate 24x7 hours a week. Failure of server system is far more catastrophic than a failure of desktop. Enterprise will lose revenue if

the server is unavailable.

•**Scalability:** as the business grows, the server may have to provide more functionality/ services. Thus ability to scale up the computin g capacity, memory, storage and I/O bandwidth is crucial.

•**Throughput**: transactions completed per minute or web pages served per second are crucial for servers.

Embedded Computers

Simple embedded microprocessors are seen in washing machines, printers, network switches, handheld devices such as cell phones, smart cards video game devices etc. embedded computers have the widest spread of processing power and cost. The primary goal is often meeting the performance need at a minimum price rather than achieving higher performance at a higher price. The other two characteristic requirements are to minimize the memory and power.

In many embedded applications, the memory can be substantial portion of the systems cost and it is very important to optimize the memory size in such cases. The application is expected to fit totally in the memory on the p rocessor chip or off chip memory. The importance of memory size translates to an emphasis on code size which is dictated by the application. Larger memory consumes more power. All these aspects are considered while choosing or designing processor for the embedded applications.

9. Calculate FIT and MTTF for 10 disks (1M hour MTTF per disk), 1 disk controller (0.5M hour MTTF), and 1 power supply (0.2M hour MTTF) . (June/July13, Jan 14)

$$\begin{aligned} FailureRate &= 10 \times (1/1{,}000{,}000) + 1/500{,}000 + 1/200{,}000 \\ &= 10 + 2 + 5/1{,}000{,}000 \\ &= 17/1{,}000{,}000 \\ &= 17{,}000 FIT \\ MTTF &= 1{,}000{,}000{,}000/17{,}000 \\ &\approx 59{,}000 hours \end{aligned}$$

10. We will run two application needs 80% of the resources and the other only 20% of the resources.
i>Given that 40% of the first application is parallelizable, how much speed up would you achieve with that application if run in isolation?
ii>Given that 99%pf the second application is parallelized, how much speed up would this application observe if run in isolation?
iii> Given that 40% of the first application is parallelizable, how much overall speed up would you observe if you parallelized it?

iv> Given that 99%pf the second application is parallelized, how much overall speed

up would you get? (June2013)(June 2016)

$$\text{Execution time}_{new} = \text{Execution time}_{old} \times \left[(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction enhanced}}{\text{Speed up}_{enhanced}} \right]$$

$$\text{Speedup}_{overall} = \frac{\text{Execution time}_{old}}{\text{Execution time}_{new}}$$

UNIT 2

1. With a neat diagram explain the classic five stage pipeline for a RISC processor. (June 2014) (June 2013) (Jan 2015)(June 2016)

Instruction set of implementation in RISC takes at most 5 cycles without pipelining. The 5 clock cycles are:

1. Instruction fetch (IF) cycle:

Send the content of program count (PC) to memory and fetch the current instruction from memory to update the PC.

New PC ← [PC] + 4; Since each instruction is 4 bytes

2. Instruction decode / Register fetch cycle (ID):

Decode the instruction and access the register file. Decoding is done in parallel with reading registers, which is possible because the register specifies are at a fixed location in a RISC architecture. This corresponds to fixed field decoding. In addition it involves:

- Perform equality test on the register as they are read for a possible branch.
- Sign-extend the offset field of the instruction in case it is needed.
- Compute the possible branch target address.

3. Execution / Effective address Cycle (EXE)

The ALU operates on the operands prepared in the previous cycle and performs one of the following function defending on the instruction type.

* Memory reference: Effective address ← [Base Register] + offset

* Register- Register ALU instruction: ALU performs the operation specified in the instruction using the values read from the register file.
* Register- Immediate ALU instruction: ALU performs the operation specified in the instruction using the first value read from the register file and that sign extended immediate.

4. Memory access (MEM)

For a load instruction, using effective address the memory is read. For a store instruction memory writes the data from the 2nd register read using effective address.

5. Write back cycle (WB)

Write the result in to the register file, whether it comes from memory system (for a LOAD instruction) or from the ALU.

Each instruction taken at most 5 clock cycles for the execution
* Instruction fetch cycle (IF)
* Instruction decode / register fetch cycle (ID)
* Execution / Effective address cycle (EX)
* Memory access (MEM)
* Write back cycle (WB)

The execution of the instruction comprising of the above subtask can be pipelined. Each of the clock cycles from the previous section becomes a pipe stage – a cycle in the pipeline. A new instruction can be started on each clock cycle which results in the execution pattern shown figure 2.1. Though each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions as illustrated in figure 2.1.

Instruction #	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				IF	ID	EXE	MEM	WB	
Instruction I+4					IF	ID	EXE	MEM	WB

Figure 2.1 Simple RISC Pipeline. On each clock cycle another instruction fetched

Each stage of the pipeline must be independent of the other stages. Also, two different operations can't be performed with the same data path resource on the same clock. For example, a single ALU cannot be used to compute the effective address and perform a subtract operation during the same clock cycle. An adder is to be provided in the stage 1 to compute new PC value and an ALU in the stage 3 to perform the arithmetic indicatedin the instruction (See figure 2.2). Conflict should not arise out of overlap of instructions using pipeline. In other words, functional unit of each stage need to be independent of other functional unit. There are three observations due to which the risk of conflict is reduced.

• Separate Instruction and data memories at the level of L1 cache eliminates a conflict for a single memory that would arise between instruction fetch and data access.
• Register file is accessed during two stages namely ID stage WB. Hardware should allow to perform maximum two reads one write every clock cycle.
• To start a new instruction every cycle, it is necessary to increment and store the PC every cycle.

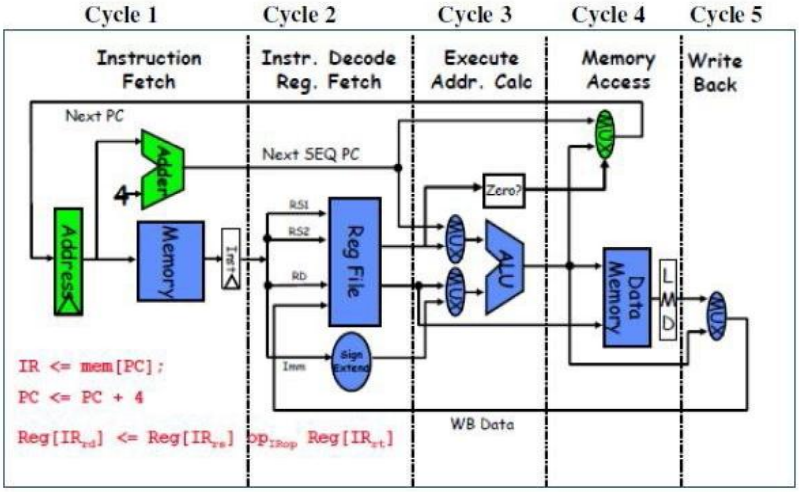


Figure 2.2 Diagram indicating the cycle and functional unit of each stage.

Buffers or registers are introduced between successive stages of the pipeline so that at the end of a clock cycle the results from one stage are stored into a register (see figure 2.3). During the next clock cycle, the next stage will use the content of these buffers as input. Figure 2.4 visualizes the pipeline activity.

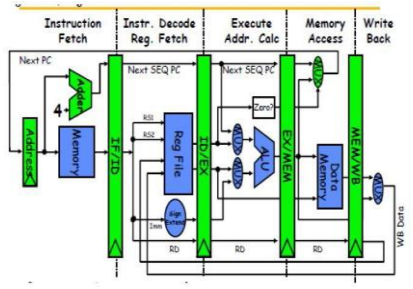


Figure 2.3 Functional units of 5 stage Pipeline. IF-ID is a buffer between IF and ID stage.

2. What are the major hurdles of pipelining? Illustrate the branch hazard in detail? (June 2014) (July 2013) (Jan 2014)(Jan 2016)

Hazards may cause the pipeline to stall. When an instruction is stalled, all the instructions issued later than the stalled instructions are also stalled. Instructions issued earlier than the stalled instructions will continue in a normal way. No new instructions are fetched during the stall. Hazard is situation that prevents the next instruction in the instruction stream fromk executing during its designated clock cycle. Hazards will reduce the pipeline performance.

Performance with Pipeline stall

A stall causes the pipeline performance to degrade from ideal performance. Performance improvement from pipelining is obtained from:

$$\textbf{Speedup} = \frac{\text{Average instruction time un-pipelined}}{\text{Average instruction time pipelined}}$$

$$\textbf{Speedup} = \frac{\text{CPI unpipelined} * \text{Clock cycle unpipelined}}{\text{CPI pipelined} * \text{Clock cycle pipelined}}$$

$$\textbf{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$\textbf{CPI pipelined} = 1 + \text{Pipeline stall clock cycles per instruction}$$

Assume that,
i) cycle time overhead of pipeline is ignored
ii) stages are balanced
With theses assumptions

$$\text{Clock cycle unpipelined} = \text{clock cycle pipelined}$$

$$\text{Therefore, Speedup} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}}$$

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1+\text{Pipeline stall cycles per instruction}}$$

If all the instructions take the same number of cycles and is equal to the number of pipeline stages or depth of the pipeline, then,

$$\text{CPI unpipelined} = \text{Pipeline depth}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1+\text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls,
Pipeline stall cycles per instruction = zero
Therefore,

Speedup = Depth of the pipeline.

When a branch is executed, it may or may not change the content of PC. If a branch is taken, the content of PC is changed to target address. If a branch is taken, the content of PC is not changed

The simple way of dealing with the branches is to redo the fetch of the instruction following a branch. The first IF cycle is essentially a stall, because, it never performs useful work. One stall cycle for every branch will yield a performance loss 10% to 30% depending on the branch frequency

Reducing the Brach Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay

1. Freeze or Flush the pipeline, holding or deleting any instructions after the ranch until the branch destination is known. It is a simple scheme and branch penalty is fixed and cannot be reduced by software

2. Treat every branch as not taken, simply allowing the hardware to continue as if the branch were not to executed. Care must be taken not to change the processor state until the branch outcome is known.

Instructions were fetched as if the branch were a normal instruction. If the branch is taken, it is necessary to turn the fetched instruction in to a no-of instruction and restart the fetch at the target address. Figure 2.8 shows the timing diagram of both the situations.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Untaken Branch	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				IF	ID	EXE	MEM	WB	
Instruction I+4					IF	ID	EXE	MEM	WB
Taken Branch	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	Idle	Idle	Idle	Idle	Idle		
Branch Target			IF	ID	EXE	MEM	WB		
Branch Target+1				IF	ID	EXE	MEM	WB	
Branch Target+2					IF	ID	EXE	MEM	WB

Figure 2.8 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom).

3. Treat every branch as *taken*: As soon as the branch is decoded and target Address is computed, begin fetching and executing at the target if the branch target is known before branch outcome, then this scheme gets advantage.

For both predicated taken or predicated not taken scheme, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware choice.

4. Delayed branch technique is commonly used in early RISC processors.

In a delayed branch, the execution cycle with a branch delay of one is

Branch instruction
Sequential successor-1
Branch target if taken

The sequential successor is in the branch delay slot and it is executed irrespective of whether or not the branch is taken. The pipeline behavior with a branch delay is shown in Figure 2.9. Processor with delayed branch, normally have a single instruction delay. Compiler has to make the successor instructions valid and useful there are three ways in which the to delay slot can be filled by the compiler.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Untaken Branch	IF	ID	EXE	MEM	WB				
Branch delay Instruction (i+1)		IF	ID	EXE	MEM	WB			
Instruction (i+2)			IF	ID	EXE	MEM	WB		
Instruction (i+3)				IF	ID	EXE	MEM	WB	
Instruction (i+4)					IF	ID	EXE	MEM	WB
Taken Branch	IF	ID	EXE	MEM	WB				
Branch delay Instruction (i+1)		IF	ID	EXE	MEM	WB			
Branch Target			IF	ID	EXE	MEM	WB		
Branch Target+1				IF	ID	EXE	MEM	WB	
Branch Target+2					IF	ID	EXE	MEM	WB

Figure 2.9 Timing diagram of the pipeline to show the behavior of a delayed branch is the same whether or not the branch is taken.

The limitations on delayed branch arise from

i) Restrictions on the instructions that are scheduled in to delay slots.
ii) Ability to predict at compiler time whether a branch is likely to be taken or not taken.

The delay slot can be filled from choosing an instruction

a) From before the branch instruction
b) From the target address
c) From fall- through path.

The principle of scheduling the branch delay is shown in fig 2.10

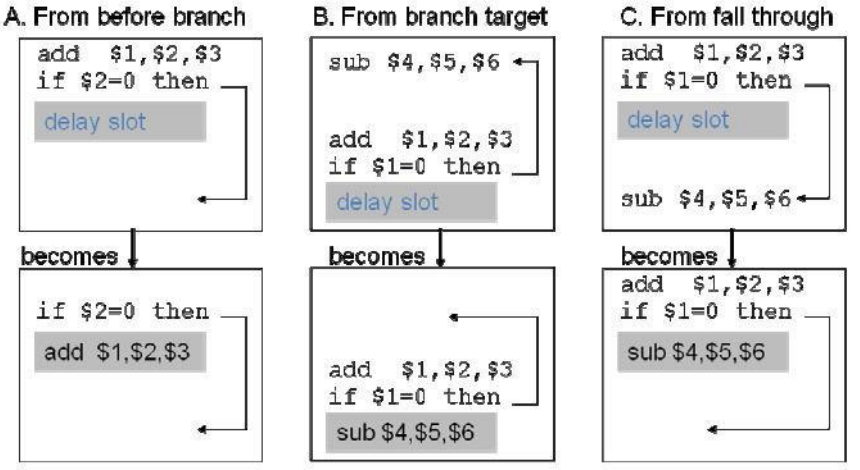


Figure 2.10 Scheduling the Branch delay

3 With a neat diagram explain the classic five stage pipeline for a RISC processor. (June 2013) (June 2015)

Instruction set of implementation in RISC takes at most 5 cycles without pipelining.
The 5 clock cycles are:

1. Instruction fetch (IF) cycle:

Send the content of program count (PC) to memory and fetch the current instruction from memory to update the PC.

New PC ← [PC] + 4; Since each instruction is 4 bytes

2. Instruction decode / Register fetch cycle (ID):

Decode the instruction and access the register file. Decoding is done in parallel with reading registers, which is possible because the register specifies are at a fixed location in a RISC architecture. This corresponds to fixed field decoding. In addition it involves:

- Perform equality test on the register as they are read for a possible branch.
- Sign-extend the offset field of the instruction in case it is needed.
- Compute the possible branch target address.

3. Execution / Effective address Cycle (EXE)

The ALU operates on the operands prepared in the previous cycle and performs one of the following function defending on the instruction type.

* Memory reference: Effective address ← [Base Register] + offset

* Register- Register ALU instruction: ALU performs the operation specified in the instruction using the values read from the register file.
* Register- Immediate ALU instruction: ALU performs the operation specified in the instruction using the first value read from the register file and that sign extended immediate.

4. Memory access (MEM)

For a load instruction, using effective address the memory is read. For a store instruction memory writes the data from the 2nd register read using effective address.

5. Write back cycle (WB)

Write the result in to the register file, whether it comes from memory system (for a LOAD instruction) or from the ALU.

Each instruction taken at most 5 clock cycles for the execution

* Instruction fetch cycle (IF)
* Instruction decode / register fetch cycle (ID)
* Execution / Effective address cycle (EX)
* Memory access (MEM)
* Write back cycle (WB)

Instruction #	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				IF	ID	EXE	MEM	WB	
Instruction I+4					IF	ID	EXE	MEM	WB

Figure 2.1 Simple RISC Pipeline. On each clock cycle another instruction fetched

Each stage of the pipeline must be independent of the other stages. Also, two different operations can't be performed with the same data path resource on the same clock. For example, a single ALU cannot be used to compute the effective address and perform a subtract operation during the same clock cycle. An adder is to be provided in the stage 1 to compute new PC value and an ALU in the stage 3 to perform the arithmetic indicatedin the instruction (See figure 2.2). Conflict should not arise out of overlap of instructions using pipeline. In other words, functional unit of each stage need to be independent of other functional unit. There are three observations due to which the risk of conflict is reduced.

• Separate Instruction and data memories at the level of L1 cache eliminates a conflict for a single memory that would arise between instruction fetch and data access.

• Register file is accessed during two stages namely ID stage WB. Hardware should allow to perform maximum two reads one write every clock cycle.

• To start a new instruction every cycle, it is necessary to increment and store the PC every cycle.

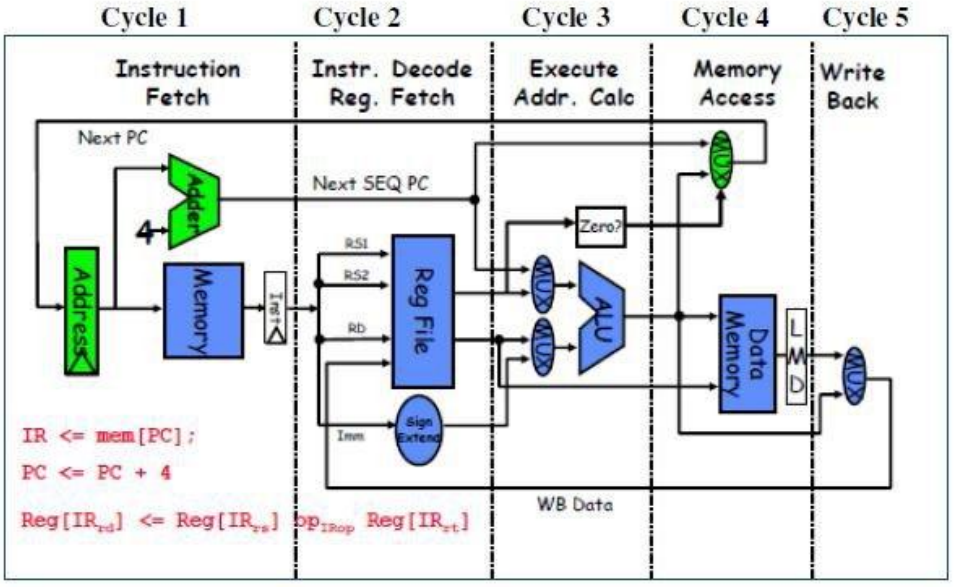


Figure 2.2 Diagram indicating the cycle and functional unit of each stage.

Buffers or registers are introduced between successive stages of the pipeline so that at the end of a clock cycle the results from one stage are stored into a register (see figure 2.3). During the next clock cycle, the next stage will use the content of these buffers as input. Figure 2.4 visualizes the pipeline activity.

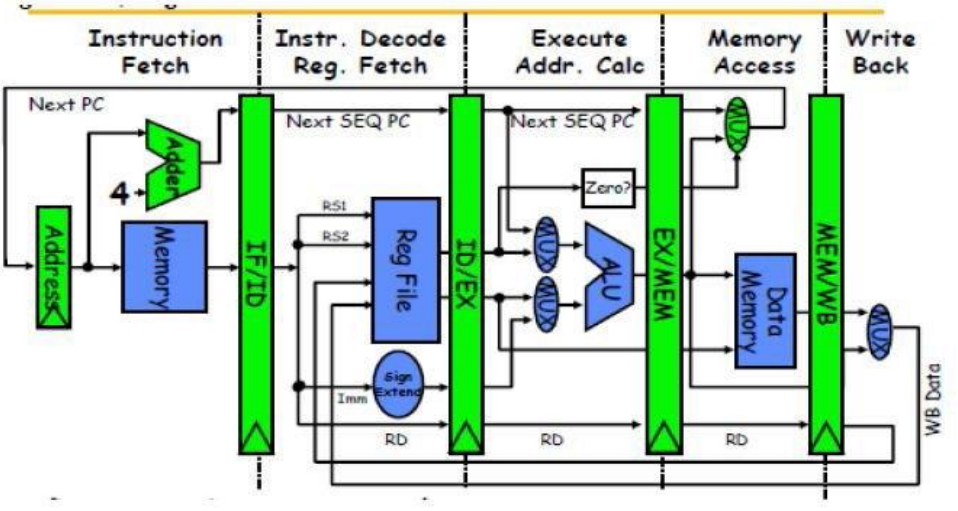


Figure 2.3 Functional units of 5 stage Pipeline. IF/ID is a buffer between IF and ID stage.

4. Explain how pipeline is implemented in MIPS. (Dec 2014) (June 2015)

Instruction set of implementation in RISC takes at most 5 cycles without pipelining. The 5 clock cycles are:

1. Instruction fetch (IF) cycle:

Send the content of program count (PC) to memory and fetch the current instruction from memory to update the PC.

New PC ← [PC] + 4; Since each instruction is 4 bytes

2. Instruction decode / Register fetch cycle (ID):

Decode the instruction and access the register file. Decoding is done in parallel with reading registers, which is possible because the register specifies are at a fixed location in a RISC architecture. This corresponds to fixed field decoding. In addition it involves:

- Perform equality test on the register as they are read for a possible branch.
- Sign-extend the offset field of the instruction in case it is needed.
- Compute the possible branch target address.

3. Execution / Effective address Cycle (EXE)

The ALU operates on the operands prepared in the previous cycle and performs one of the following function defending on the instruction type.

* Memory reference: Effective address ← [Base Register] + offset

* Register- Register ALU instruction: ALU performs the operation specified in the instruction using the values read from the register file.
* Register- Immediate ALU instruction: ALU performs the operation specified in the instruction using the first value read from the register file and that sign extended immediate.

4. Memory access (MEM)

For a load instruction, using effective address the memory is read. For a store instruction memory writes the data from the 2nd register read using effective address.

5. Write back cycle (WB)

Write the result in to the register file, whether it comes from memory system (for a LOAD instruction) or from the ALU.

Each instruction taken at most 5 clock cycles for the execution

* Instruction fetch cycle (IF)
* Instruction decode / register fetch cycle (ID)
* Execution / Effective address cycle (EX)
* Memory access (MEM)
* Write back cycle (WB)

The execution of the instruction comprising of the above subtask can be pipelined. Each of the clock cycles from the previous section becomes a pipe stage – a cycle in the pipeline. A new instruction can be started on each clock cycle which results in the execution pattern shown figure 2.1. Though each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions as illustrated in figure 2.1.

Write the result in to the register file, whether it comes from memory system (for a LOAD instruction) or from the ALU.

Each instruction taken at most 5 clock cycles for the execution
* Instruction fetch cycle (IF)
* Instruction decode / register fetch cycle (ID)
* Execution / Effective address cycle (EX)
* Memory access (MEM)
* Write back cycle (WB)

The execution of the instruction comprising of the above subtask can be pipelined. Each of the clock cycles from the previous section becomes a pipe stage – a cycle in the pipeline. A new instruction can be started on each clock cycle which results in the execution pattern shown figure 2.1. Though each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions as illustrated in figure 2.1.

Instruction #	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				IF	ID	EXE	MEM	WB	
Instruction I+4					IF	ID	EXE	MEM	WB

Figure 2.1 Simple RISC Pipeline. On each clock cycle another instruction fetched

Each stage of the pipeline must be independent of the other stages. Also, two different operations can't be performed with the same data path resource on the same clock. For example, a single ALU cannot be used to compute the effective address and perform a subtract operation during the same clock cycle. An adder is to be provided in the stage 1 to compute new PC value and an ALU in the stage 3 to perform the arithmetic indicatedin the instruction (See figure 2.2). Conflict should not arise out of overlap of instructions using pipeline. In other words, functional unit of each stage need to be independent of other functional unit. There are three observations due to which the risk of conflict is reduced.

• Separate Instruction and data memories at the level of L1 cache eliminates a conflict for a single memory that would arise between instruction fetch and data access.

• Register file is accessed during two stages namely ID stage WB. Hardware should allow to perform maximum two reads one write every clock cycle.

• To start a new instruction every cycle, it is necessary to increment and store the PC every cycle.

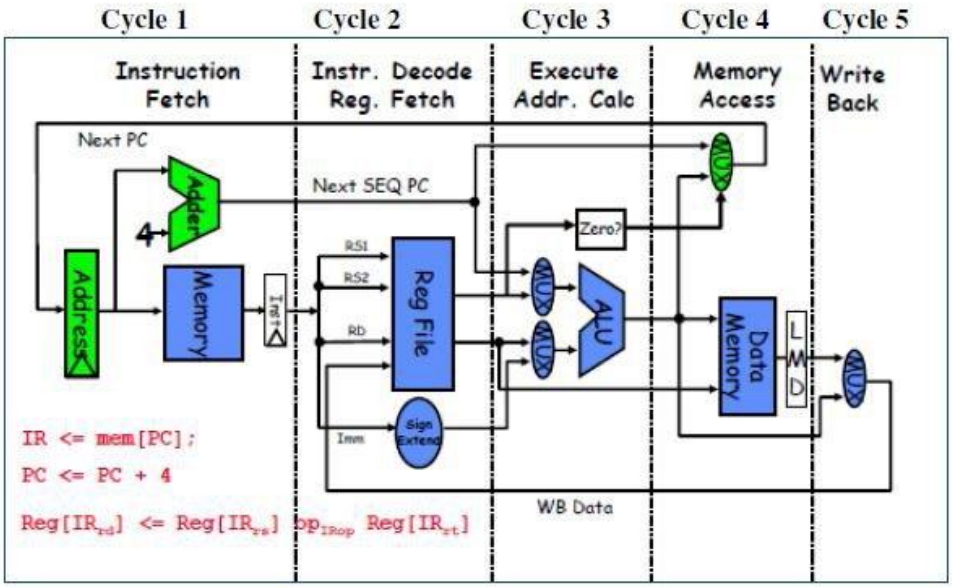


Figure 2.2 Diagram indicating the cycle and functional unit of each stage

Buffers or registers are introduced between successive stages of the pipeline so that at the end of a clock cycle the results from one stage are stored into a register (see figure 2.3). During the next clock cycle, the next stage will use the content of these buffers as input. Figure 2.4 visualizes the pipeline activity.

5. Explain different techniques in reducing pipeline branch penalties. (Dec 2014) (June2013) (Jan 2015)

Reducing the Brach Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay

1. Freeze or Flush the pipeline, holding or deleting any instructions after the ranch until the branch destination is known. It is a simple scheme and branch penalty is fixed and cannot be reduced by software

2. Treat every branch as not taken, simply allowing the hardware to continue as if the branch were not to executed. Care must be taken not to change the processor state until the branch outcome is known.

Instructions were fetched as if the branch were a normal instruction. If the branch is taken, it is necessary to turn the fetched instruction in to a no-of instruction and restart the fetch at the target address. Figure 2.8 shows the timing diagram of both the situations.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Untaken Branch	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				IF	ID	EXE	MEM	WB	
Instruction I+4					IF	ID	EXE	MEM	WB
Taken Branch	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	Idle	Idle	Idle	Idle	Idle		
Branch Target			IF	ID	EXE	MEM	WB		
Branch Target+1				IF	ID	EXE	MEM	WB	
Branch Target+2					IF	ID	EXE	MEM	WB

Figure 2.8 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom).

3. Treat every branch as *taken*: As soon as the branch is decoded and target Address is computed, begin fetching and executing at the target if the branch target is known before branch outcome, then this scheme gets advantage.

For both predicated taken or predicated not taken scheme, the compiler can improve performance by organizing the code so that the most frequent path

matches the hardware choice.

4. Delayed branch technique is commonly used in early RISC processors.

In a delayed branch, the execution cycle with a branch delay of one is
Branch instruction
Sequential successor-1
Branch target if taken

The sequential successor is in the branch delay slot and it is executed irrespective of whether or not the branch is taken. The pipeline behavior with a branch delay is shown in Figure 2.9. Processor with delayed branch, normally have a single instruction delay. Compiler has to make the successor instructions valid and useful there are three ways in which the to delay slot can be filled by the compiler.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Untaken Branch	IF	ID	EXE	MEM	WB				
Branch delay Instruction (i+1)		IF	ID	EXE	MEM	WB			
Instruction (i+2)			IF	ID	EXE	MEM	WB		
Instruction (i+3)				IF	ID	EXE	MEM	WB	
Instruction (i+4)					IF	ID	EXE	MEM	WB
Taken Branch	IF	ID	EXE	MEM	WB				
Branch delay Instruction (i+1)		IF	ID	EXE	MEM	WB			
Branch Target			IF	ID	EXE	MEM	WB		
Branch Target+1				IF	ID	EXE	MEM	WB	
Branch Target+2					IF	ID	EXE	MEM	WB

Figure 2.9 Timing diagram of the pipeline to show the behavior of a delayed branch is the same whether or not the branch is taken.

The limitations on delayed branch arise from

i) Restrictions on the instructions that are scheduled in to delay slots.

ii) Ability to predict at compiler time whether a branch is likely to be taken or not taken.

The delay slot can be filled from choosing an instruction

a) From before the branch instruction

b) From the target address

c) From fall- through path.

The principle of scheduling the branch delay is shown in fig 2.10

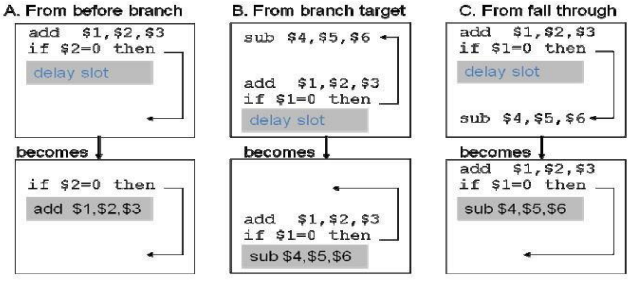


Figure 2.10 Scheduling the Branch delay

6.What are the major hurdles of pipelining? Explain briefly. (July 2014) (Dec 2013)

Pipeline Hazards

Hazards may cause the pipeline to stall. When an instruction is stalled, all the instructions issued later than the stalled instructions are also stalled. Instructions issued earlier than the stalled instructions will continue in a normal way. No new instructions are fetched during the stall. Hazard is situation that prevents the next instruction in the instruction stream fromk executing during its designated clock cycle. Hazards will reduce the pipeline performance.

Performance with Pipeline stall

A stall causes the pipeline performance to degrade from ideal performance. Performance improvement from pipelining is obtained from:

$$\textbf{Speedup} = \frac{\text{Average instruction time un-pipelined}}{\text{Average instruction time pipelined}}$$

$$\textbf{Speedup} = \frac{\text{CPI unpipelined} * \text{Clock cycle unpipelined}}{\text{CPI pipelined} * \text{Clock cycle pipelined}}$$

$$\textbf{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$\textbf{CPI pipelined} = 1 + \text{Pipeline stall clock cycles per instruction}$$

Assume that,
i) cycle time overhead of pipeline is ignored
ii) stages are balanced
With theses assumptions

$$\text{Clock cycle unpipelined} = \text{clock cycle pipelined}$$
$$\text{Therefore, Speedup} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}}$$

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1+\text{Pipeline stall cycles per instruction}}$$

If all the instructions take the same number of cycles and is equal to the number of pipeline stages or depth of the pipeline, then,

$$\text{CPI unpipelined} = \text{Pipeline depth}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1+\text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls,
Pipeline stall cycles per instruction = zero
Therefore,
Speedup = Depth of the pipeline.

7. List and explain five ways of classifying exception in a computer system. (July 2013) (Jan 2015)(Jan 2016)

Types of exceptions:

The term exception is used to cover the terms interrupt, fault and exception. I/O device request, page fault, Invoking an OS service from a user program, Integer arithmetic overflow, memory protection overflow, Hardware malfunctions, Power failure etc. are the different classes of exception. Individual events have important characteristics that determine what action is needed corresponding to that exception.

i) Synchronous versus Asynchronous

If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is asynchronous. Asynchronous events are caused by devices external to the CPU and memory such events are handled after the completion of the current instruction.

ii) User requested versus coerced:

User requested exceptions are predictable and can always be handled after the current instruction has completed. Coerced exceptions are caused by some hardware event that is not under the control of the user program. Coerced exceptions are harder to implement because they are not predictable

iii) User maskable versus user non maskable :

If an event can be masked by a user task, it is user maskable. Otherwise it is user non maskable.

iv) Within versus between instructions:

Exception that occur within instruction are usually synchronous, since the instruction triggers the exception. It is harder to implement exceptions that occur withininstructions than those between instructions, since the instruction must be stopped and restarted. Asynchronous exceptions that occurs within instructions arise from catastrophic situations and always causes program termination.

v) Resume versus terminate:

If the program's execution continues after the interrupt, it is a resuming event otherwise if is terminating event. It is easier implement exceptions that terminate execution. 29

8. List pipeline hazards. Explain any one in detail. (June2013) (June 2015)(June 2016)

Hazards may cause the pipeline to stall. When an instruction is stalled, all the instructions issued later than the stalled instructions are also stalled. Instructions issued earlier than the stalled instructions will continue in a normal way. No new instructions are fetched during the stall. Hazard is situation that prevents the next instruction in the instruction stream fromk executing during its designated clock cycle. Hazards will reduce the pipeline performance.

Performance with Pipeline stall

A stall causes the pipeline performance to degrade from ideal performance. Performance improvement from pipelining is obtained from:

$$\textbf{Speedup} = \frac{\text{Average instruction time un-pipelined}}{\text{Average instruction time pipelined}}$$

$$\textbf{Speedup} = \frac{\text{CPI unpipelined} * \text{Clock cycle unpipelined}}{\text{CPI pipelined} * \text{Clock cycle pipelined}}$$

$$\textbf{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$\textbf{CPI pipelined} = 1 + \text{Pipeline stall clock cycles per instruction}$$

Assume that,
i) cycle time overhead of pipeline is ignored
ii) stages are balanced
With theses assumptions

$$\text{Clock cycle unpipelined} = \text{clock cycle pipelined}$$

$$\text{Therefore, Speedup} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}}$$

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1+\text{Pipeline stall cycles per instruction}}$$

If all the instructions take the same number of cycles and is equal to the number of pipeline stages or depth of the pipeline, then,

$$\text{CPI unpipelined} = \text{Pipeline depth}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1+\text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls,
Pipeline stall cycles per instruction = zero
Therefore,
Speedup = Depth of the pipeline.

When a branch is executed, it may or may not change the content of PC. If a branch is taken, the content of PC is changed to target address. If a branch is taken, the content of PC is not changed

The simple way of dealing with the branches is to redo the fetch of the instruction following a branch. The first IF cycle is essentially a stall, because, it never performs useful work. One stall cycle for every branch will yield a performance loss 10% to 30% depending on the branch frequency

UNIT 3

1. What are the techniques used to reduce branch costs? Explain both static and dynamic branch prediction used for same? (June 2014) (June2013) (June 2015)(June 2016)

To keep a pipe line full, parallelism among instructions must be exploited by finding sequence of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall,a dependent instruction must be separated from the source instruction by the distance in clock cycles equal to the pipeline latency of that source instruction. A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline.

The compiler can increase the amount of available ILP by transferring loops.

```
for(i=1000; i>0 ;i=i-1)
X[i] = X[i] + s;
```

We see that this loop is parallel by the noticing that body of the each iteration is independent.

The first step is to translate the above segment to MIPS assembly language

```
Loop: L.D F0, 0(R1) : F0=array element

ADD.D F4, F0, F2 : add scalar in F2
S.D F4, 0(R1) : store result
DADDUI R1, R1, #-8 : decrement pointer
: 8 Bytes (per DW)
BNE R1, R2, Loop : branch R1! = R2
```

Without any Scheduling the loop will execute as follows and takes 9 cycles for each iteration.

1 Loop: L.D F0, 0(R1) ;F0=vector element
2 stall
3 ADD.D F4, F0, F2 ;add scalar in F2
4 stall
5 stall
6 S.D F4, 0(R1) ;store result
7 DADDUI R1, R1,# -8 ;decrement pointer 8B (DW)
8 stall ;assumes can't forward to branch
9 BNEZ R1, Loop ;branch R1!=zero

We can schedule the loop to obtain only two stalls and reduce the time to 7 cycles:

L.D F0, 0(R1)

DADDUI R1, R1, #-8
ADD.D F4, F0, F2

Stall

Stall

S.D F4, 0(R1)

BNE R1, R2, Loop

Loop Unrolling can be used to minimize the number of stalls. Unrolling the body of the loop by our times, the execution of four iteration can be done in 27 clock cycles or 6.75 clock cycles per iteration.

1 Loop: L.D F0,0(R1)

3 ADD.D F4,F0,F2

6 S.D 0(R1),F4 ;drop DSUBUI & BNEZ

7 L.D F6,-8(R1)

9 ADD.D F8,F6,F2

12 S.D -8(R1),F8 ;drop DSUBUI & BNEZ

13 L.D F10,-16(R1)

15 ADD.D F12,F10,F2

18 S.D -16(R1),F12 ;drop DSUBUI & BNEZ

19 L.D F14,-24(R1)

21 ADD.D F16,F14,F2

24 S.D -24(R1),F16

25 DADDUI R1,R1,#-32 :alter to 4*8

26 BNEZ R1,LOOP

Unrolled loop that minimizes the stalls to 14 clock cycles for four iterations is given below:

1 Loop: L.D F0, 0(R1)

2 L.D F6, -8(R1)

3 L.D F10, -16(R1)

4 L.D F14, -24(R1)
5 ADD.D F4, F0, F2

6 ADD.D F8, F6, F2

7 ADD.D F12, F10, F2

8 ADD.D F16, F14, F2

9 S.D 0(R1), F4

10 S.D -8(R1), F8

11 S.D -16(R1), F12

12 DSUBUI R1, R1,#32

13 S.D 8(R1), F16 ;8-32 = -24

14 BNEZ R1, LOOP

Summary of Loop unrolling and scheduling

The loop unrolling requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:

1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)

2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code

4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent

• Transformation requires analyzing memory addresses and finding that they do not refer to the same address

5. Schedule the code, preserving any dependences needed to yield the same result as the original code

To reduce the Branch cost, prediction of the outcome of the branch may be done. The prediction may be done statically at compile time using compiler support or dynamically using hardware support. Schemes to reduce the impact of control hazard are discussed below:

Static Branch Prediction

Assume that the branch will not be ***taken*** and continue execution down the sequential instruction stream. If the branch is ***taken***, the instruction that are being fetched and decoded must be discarded. Execution continues at the branch target. Discarding instructions means we must be able to flush instructions in the IF, ID and EXE stages.
Alternately, it is possible that the branch can be predicted as taken. As soon as the instruction decoded is found as branch, at the earliest, start fetching the instruction from the target address.

Dynamic Branch Prediction

With deeper pipelines the branch penalty increases when measured in clock cycles. Similarly, with multiple issue, the branch penalty increases in terms of instructions lost. Hence, a simple static prediction scheme is inefficient or may not be efficient in most of the situations. One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and if so, to begin fetching new instruction from the target address.
This technique is called ***Dynamic branch prediction***.
• Why does prediction work?

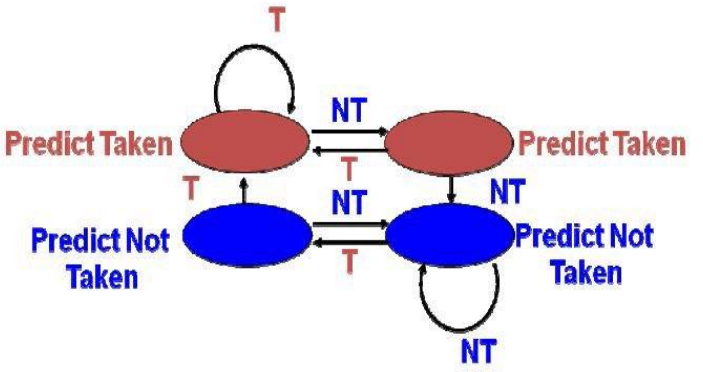
– Underlying algorithm has regularities

– Data that is being operated on has regularities
– Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems.
– There are a small number of important branches in programs which have dynamic behavior for which dynamic branch prediction performance will be definitely better compared to static branch prediction.

• Performance = *f*(accuracy, cost of misprediction)
• Branch History Table (BHT) is used to dynamically predict the outcome of the current branch instruction. Lower bits of PC address index table of 1-bit values
– Says whether or not branch taken last time
- No address check

• Problem: in a loop, 1-bit BHT will cause two mispredictions (average is 9 iterations before exit):
– End of loop case, when it exits instead of looping as before
– First time through loop on *next* time through code, when it predicts exit instead of looping

• Simple two bit history table will give better performance. The four different states of 2 bit predictor is shown in the state transition diagram.



The states in a 2-bit prediction scheme

Correlating Branch Predictor

It may be possible to improve the prediction accuracy by considering the recent behavior of other branches rather than just the branch under consideration. Correlating predictors are two-level predictors. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch.

• Idea: record *m* most recently executed branches as taken or not taken, and use that pattern to select the proper *n*-bit branch history table (BHT)

• In general, (*m*,*n*) predictor means record last *m* branches to select between 2m history tables, each with *n*-bit counters

– Thus, old 2-bit BHT is a (0,2) predictor

– Global Branch History: *m*-bit shift register keeping T/NT status of last *m* branches.

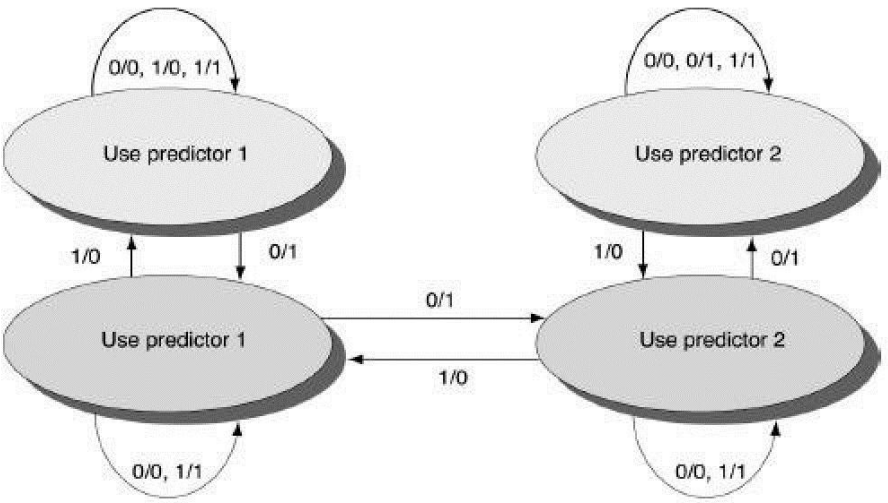
• Each entry in table has *m n*-bit predictors. In case of (2,2) predictor, behavior of recent branches selects between four predictions of next branch, updating just that prediction. The scheme of the table is shown

Tournament predictor is a multi level branch predictor and uses n bit saturating counter to chose between predictors. The predictors used are global predictor and local predictor.

– Advantage of tournament predictor is ability to select the right predictor for a particular branch which is particularly crucial for integer benchmarks.

– A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks

– Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor was most effective oin recent prediction.



Dynamic Branch Prediction Summary

• Prediction is becoming important part of execution as it improves the performance of the pipeline.

• Branch History Table: 2 bits for loop accuracy

• Correlation: Recently executed branches correlated with next branch
  – Either different branches (GA)
  – Or different executions of same branches (PA)
• Tournament predictors take insight to next level, by using multiple predictors
  – usually one based on global information and one based on local information, and combining them with a selector

2. with a neat diagram give the basic structure of tomasulo based MIPS FP unit and explain the various field of reservation stations . (June 2014) (Dec 2013) (Jan 2015) (June 2015)(Jan 2016)

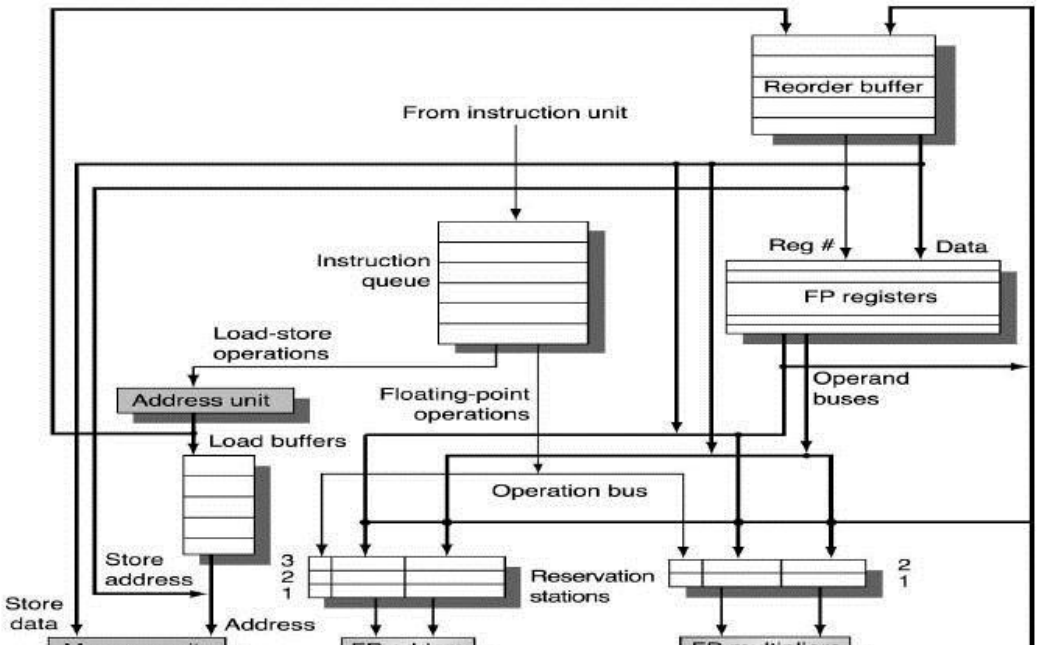
Tomasulu algorithm and Reorder Buffer

Tomasulu idea:

1. Have reservation stations where register renaming is possible
2. Results are directly forwarded to the reservation station along with the final registers. This is also called short circuiting or bypassing.

ROB:

1.The instructions are stored sequentially but we have indicators to say if it is speculative or completed execution.
2. If completed execution and not speculative and reached head of the queue then we commit it.



Three components of hardware-based speculation

1. *dynamic branch prediction* to pick branch outcome
2. *speculation* to allow instructions to execute before control dependencies are resolved, i.e., before branch outcomes become known – with ability to undo in case of incorrect speculation
3. *dynamic scheduling*

Speculating with Tomasulo
Key ideas:
1. separate execution from completion: instructions to execute speculatively but no *instructions update registers or memory until no more speculative*

2. therefore, add a final step – after an instruction is no longer speculative, called *instruction commit*– when it is allowed to make register and memory updates

3. allow instructions to execute and complete out of order but force them to commit in order

4. Add hardware called the *reorder buffer* (*ROB*), with registers to hold the result of an instruction *between completion and commit*

Tomasulo's Algorithm with Speculation: Four Stages

1. Issue: get instruction from Instruction Queue
   _ if reservation station and ROB slot free (no structural hazard),
   control issues instruction to reservation station and ROB, and sends to reservation station operand values (or reservation station source for values) as well as allocated ROB slot number

2. Execution: operate on operands (EX)
_ when both operands ready then execute;if not ready, watch CDB for result

3. Write result: finish execution (WB)
_ write on CDB to all awaiting units and ROB; mark reservation station available

4. Commit: update register or memory with ROB result
   _ when instruction reaches head of ROB and results present, update register with result or store to memory and remove instruction from ROB
   _ if an incorrectly predicted branch reaches the head of ROB, flush the ROB, and restart at correct successor of branch

ROB Data Structure

ROB entry fields
• Instruction type: branch, store, register operation (i.e., ALU or load)
• State: indicates if instruction has completed and value is ready
• Destination: where result is to be written – register number for register operation (i.e. ALU or load), memory address for store
• branch has no destination result

• Value: holds the value of instruction result till time to commit
Additional reservation station field
• Destination: Corresponding ROB entry number

Example
1. L.D F6, 34(R2)

2. L.D F2, 45(R3

3. MUL.D F0, F2, F4

4. SUB.D F8, F2, F6

5. DIV.D F10, F0, F6

6. ADD.D F6, F8, F2

The position of Reservation stations, ROB and FP registers are indicated below:

Assume latencies load 1 clock, add 2 clocks, multiply 10 clocks, divide 40 clocks
Show data structures just before MUL.D goes to commit*…*

Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	yes	MUL	Mem[45+Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV		Mem[34+Regs[R2]]	#3		#5	

At the time MUL.D is ready to commit only the two L.D instructions have already

committed,though others have completed execution
Actually, the MUL.D is at the head of the ROB – the L.D instructions are shown only for understanding purposes #X represents value field of ROB entry number X

Floating point registers

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder#	3						6		4	5
Busy	yes	no	no	no	no	no	yes	…	yes	yes

Reorder Buffer

Entry	Busy	Instruction		State	Destination	Value
1	no	L.D	F6, 34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	no	L.D	F2, 45(R3)	Commit	F2	Mem[45+Regs[R3]]
3	yes	MUL.D	F0, F2, F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D	F8, F6, F2	Write result	F8	#1 – #2
5	yes	DIV.D	F10, F0, F6	Execute	F10	
6	yes	ADD.D	F6, F8, F2	Write result	F6	#4 + #2

Example

```
Loop:  LD      F0      0       R1
       MULTD   F4      F0      F2
       SD      F4      0       R1
       SUBI    R1      R1      #8
       BNEZ    R1      Loop
```

3. What are data dependencies? Explain name dependencies with examples between two instructions. (Dec 2013) (June2013) (Jan 2015)

Data Dependences

An instruction j is data dependant on instruction i if either of the following holds:

i) Instruction i produces a result that may be used by instruction j

Eg1: i: L.D **F0**, 0(R1)
j: ADD.D F4, **F0**, F2

ith instruction is loading the data into the F0 and jth instruction use F0 as one the operand. Hence, jth instruction is data dependant on ith instruction.

Eg2: DADD **R1**, R2, R3
DSUB R4, **R1**, R5

ii) Instruction j is data dependant on instruction k and instruction k data dependant on instruction i

Eg: L.D **F4**, 0(R1)
MUL.D **F0**, **F4**, F6
ADD.D F5, **F0**, F7

Dependences are the property of the programs. A Data value may flow between instructions either through registers or through memory locations. Detecting the data flow and dependence that occurs through registers is quite straight forward. Dependences that

flow through the memory locations are more difficult to detect. A data dependence convey three things.

a) The possibility of the Hazard.
b) The order in which results must be calculated and
c) An upper bound on how much parallelism can possibly exploited.

4.What are correlating predictors? Explain with examples. (June/July 2014) (Jan 2015)

Correlating Branch Predictor

It may be possible to improve the prediction accuracy by considering the recent behavior of other branches rather than just the branch under consideration. Correlating predictors are two-level predictors. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch.

• Idea: record *m* most recently executed branches as taken or not taken, and use that pattern to select the proper *n*-bit branch history table (BHT)

5. For the following instructions, using dynamic scheduling show the status of R,O.B, reservation station when only MUL.D is ready to commit and two L.D committed. (June/July 2013)(June 2016)

L.D F6, 32 (R2)
L.D F2, 44(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10,F0,F6
ADD.D F6,F8,F2
Also show the types of hazards between instructions.

ROB entry fields
• Instruction type: branch, store, register operation (i.e., ALU or load)
• State: indicates if instruction has completed and value is ready
• Destination: where result is to be written – register number for register operation (i.e. ALU or load), memory address for store
• branch has no destination result
Value: holds the value of instruction result till time to commit
Additional reservation station field
• Destination: Corresponding ROB entry number

Example
1. L.D F6, 34(R2)

2. L.D F2, 45(R3

3. MUL.D F0, F2, F4

4. SUB.D F8, F2, F6

5. DIV.D F10, F0, F6

6. ADD.D F6, F8, F2

The position of Reservation stations, ROB and FP registers are indicated below:
Assume latencies load 1 clock, add 2 clocks, multiply 10 clocks, divide 40 clocks
Show data structures just before MUL.D goes to commit*…*

Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	yes	MUL	Mem[45+Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV		Mem[34+Regs[R2]]	#3		#5	

At the time MUL.D is ready to commit only the two L.D instructions have already committed,though others have completed execution .Actually, the MUL.D is at the head of the ROB – the L.D instructions are shown only for understanding purposes #X represents value field of ROB entry number X

Floating point registers

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder#	3						6		4	5
Busy	yes	no	no	no	no	no	yes	…	yes	yes

Reorder Buffer

Entry	Busy	Instruction		State	Destination	Value
1	no	L.D	F6, 34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	no	L.D	F2, 45(R3)	Commit	F2	Mem[45+Regs[R3]]
3	yes	MUL.D	F0, F2, F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D	F8, F6, F2	Write result	F8	#1 – #2
5	yes	DIV.D	F10, F0, F6	Execute	F10	
6	yes	ADD.D	F6, F8, F2	Write result	F6	#4 + #2

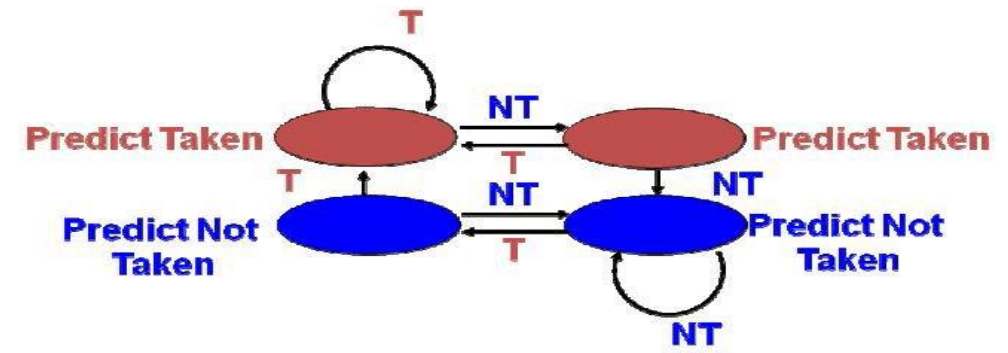
6. What is the drawback of 1- bit dynamic branch prediction method? Clearly state how it is overcome in 2- bit prediction. Give the state transition diagram of 2-bit predictor. (Jan 2014) (June2013)(Jan 2016)

There are a small number of important branches in programs which have dynamic behavior for which dynamic branch prediction performance will be definitely better compared to static branch prediction.

- Performance = *f*(accuracy, cost of misprediction)
- Branch History Table (BHT) is used to dynamically predict the outcome of the current branch instruction. Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check

- Problem: in a loop, 1-bit BHT will cause two mispredictions (average is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts exit instead of looping

- Simple two bit history table will give better performance. The four different states of 2 bit predictor is shown in the state transition diagram.



The states in a 2-bit prediction scheme

Correlating Branch Predictor

It may be possible to improve the prediction accuracy by considering the recent behavior

of other branches rather than just the branch under consideration. Correlating predictors

are two-level predictors. Existing correlating predictors add information about the

behavior of the most recent branches to decide how to predict a given branch.

• Idea: record *m* most recently executed branches as taken or not taken, and use that

pattern to select the proper *n*-bit branch history table (BHT)

• In general, (*m*,*n*) predictor means record last *m* branches to select between 2m history

tables, each with *n*-bit counters

– Thus, old 2-bit BHT is a (0,2) predictor

– Global Branch History: *m*-bit shift register keeping T/NT status of last *m*

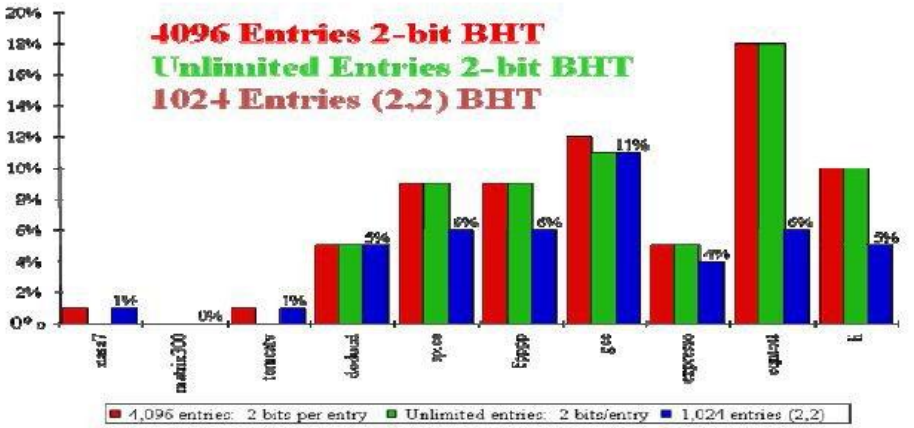
branches.

• Each entry in table has *m* *n*-bit predictors. In case of (2,2) predictor, behavior of recent

branches selects between four predictions of next branch, updating just that prediction.

The scheme of the table is shown:

Comparisons of different schemes are shown in the graph.



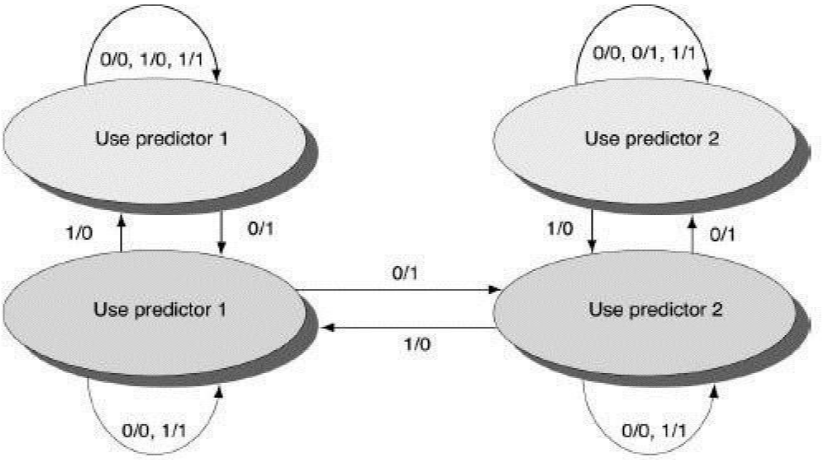
Comparison of 2 bit predictors (y-axis: % frequency of mispredictions, x-axis: SPEC89 programs)

Tournament predictor is a multi level branch predictor and uses n bit saturating counter to chose between predictors. The predictors used are global predictor and local predictor.

– Advantage of tournament predictor is ability to select the right predictor for a particular branch which is particularly crucial for integer benchmarks.

– A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks

– Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor was most effective oin recent prediction.



UNIT 4

1. Explain the basic VLIW approach for exploiting ILP using multiple issues? (June/July 2014) (June2013) (June 2015)(Jan 2016)

Exploiting ILP: Multiple Issue Computers

Multiple Issue Computers

• **Benefit**
  – CPIs go below one, use IPC instead (instructions/cycle)
  – Example: Issue width = 3 instructions, Clock = 3GHz
• Peak rate: 9 billion instructions/second, IPC = 3
• For our 5 stage pipeline, 15 instructions “in flight” at any given time
• Multiple Issue types
  – Static
• Most instruction scheduling is done by the compiler
  – Dynamic (superscalar)
• CPU makes most of the scheduling decisions

• Challenge: overcoming instruction dependencies
  – Increased latency for loads
  – Control hazards become worse
• Requires a more ambitious design
  – Compiler techniques for scheduling
  – Complex instruction decoding logic

Exploiting ILP:Multiple Issue Computers Static Scheduling

Instruction Issuing
  • Have to decide which instruction types can issue in a cycle
    – Issue packet: instructions issued in a single clock cycle
    – Issue slot: portion of an issue packet
• Compiler assumes a large responsibility for hazard checking, scheduling, etc.

Static Multiple Issue

For now, assume a "souped-up" 5-stage MIPS pipeline that can issue a packet with:
  – One slot is an ALU or branch instruction
  One slot is a load/store instruction

Instruction Type	Pipeline Stages						
ALU or Branch instruction	IF	ID	EX	MEM	WB		
Load or Store instruction	IF	ID	EX	MEM	WB		
ALU or Branch instruction		IF	ID	EX	MEM	WB	
Load or Store instruction		IF	ID	EX	MEM	WB	
ALU or Branch instruction			IF	ID	EX	MEM	WB
Load or Store instruction			IF	ID	EX	MEM	WB

–

Static Multiple Issue

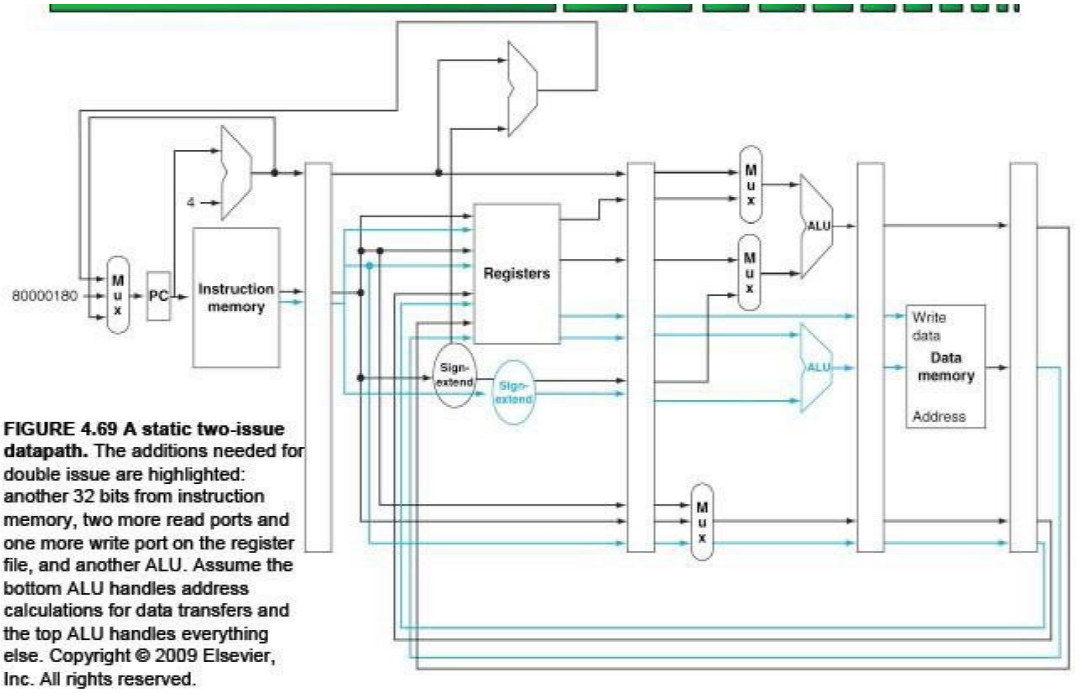


FIGURE 4.69 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

Static Multiple Issue Scheduling

```
add  $1, $2, $3
add  $5, $2, $2
load $4, $3(100)
load $3, $2(100)
sub  $2, $5, $3
add $2, $2, $4
```

Becomes

Cycle	ALU/Branch Instruction	Load/Store Instruction
1		
2		
3		
4		
5		
6		

Static Mult. Issue w/Loop Unrolling

```
Loop: lw    $t0, 0($s1)
      addu  $t0, $t0, $s2
      sw    $t0, 0($s1)
      addi  $s1, $s1, -4
      bne   $s1, $0, Loop
```

Original loop schedule for a 2-issue MIPS

	ALU/Branch	Load/Store	Cycle
Loop:			1
			2
			3
			4

Unrolled (once) loop schedule for a 2-issue MIPS

	ALU/Branch	Load/Store	Cycle
Loop:			1
			2
			3
			4
			5
			6
			7

Static Mult. Issue w/Loop Unrolling

```
Loop: lw    $t0, 0($s1)
      addu  $t0, $t0, $s2
      sw    $t0, 0($s1)
      lw    $t1, -4($s1)
      addu  $t1, $t1, $s2
      sw    $t1, -4($s1)
      addi  $s1, $s1, -8
      bne   $s1, $0, Loop
```

Exploiting ILP:Multiple Issue Computers Dynamic Scheduling

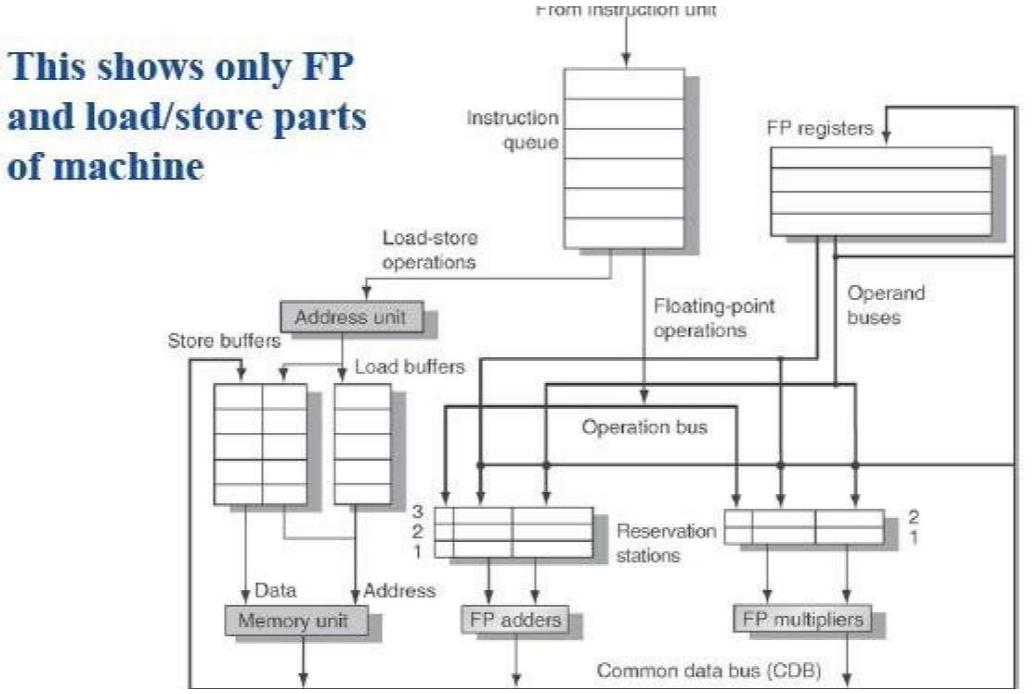
Dynamic Multiple Issue Computers
• Superscalar computers
• CPU generally manages instruction issuing and ordering
– Compiler helps, but CPU dominates
• Process
– Instructions issue in-order
– Instructions can execute out-of-order
• Execute once all operands are ready
– Instructions commit in-order
• Commit refers to when the architectural register file is updated (current completed state of program
Aside: Data Hazard Refresher
• Two instructions (i and j), j follows i in program order
• Read after Read (RAR)
• Read after Write (RAW)
– Type:
– Problem:
• Write after Read (WAR)
– Type:
– Problem:
• Write after Write (WAW)
– Type: Problem:
Superscalar Processors
• Register Renaming
– Use more registers than are defined by the architecture

Architectural registers: defined by ISA
• Physical registers: total registers
– Help with name dependencies
• Antidependence
– Write after Read hazard
• Output dependence
– Write after Write hazard

Tomasulo's Superscalar Computers

• R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM J. of Research and Development*, Jan. 1967
• See also: D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 model 91: Machine philosophy and instruction-handling," *IBM J. of Research and evelopment*, Jan. 1967

• Allows out-of-order execution
• Tracks when operands are available
– Minimizes RAW hazards
• Introduced renaming for WAW and WAR hazards
Tomasulo's Superscalar Computers



Instruction Execution Process

• Three parts, arbitrary number of cycles/part

• Above does not allow for speculative execution
• Issue (aka Dispatch)
– If empty reservation station (RS) that matches instruction, send to RS with operands rom register file and/or know which functional unit will send operand

- If no empty RS, stall until one is available

Rename registers as appropriate
Instruction Execution Process
• Execute
– All branches before instruction must be resolved
• Preserves exception behavior
– When all operands available for an instruction, send it to functional unit
• Monitor common data bus (CDB) to see if result is needed by RS entry
– For non-load/store reservation stations

• If multiple instructions ready, have to pick one to send to functional unit
– For load/store
• Compute address, then place in buffer
• Loads can execute once memory is free
• Stores must wait for value to be stored, then execute

Write Back
– Functional unit places on CDB
• Goes to both register file and reservation stations
– Use of CDB enables forwarding for RAW hazards
– Also introduces a latency between result and use of a value

2. what are the key issues in implementing advanced speculation techniques? Explain them in detail? (Jun 2014) (June2013)

Tomasulo's w/HW Speculation
• Key aspects of this design
– Separate forwarding (result bypassing) from actual instruction completion
• Assuming instructions are executing speculatively
• Can pass results to later instructions, but prevents instruction from performing updates that can't be "undone"
– Once instruction is no longer speculative it can update register file/memory
• New step in execution sequence: instruction commit
• Requires instructions to wait until they can commit Commits still happen in order
Reorder Buffer (ROB)

Instructions hang out here before committing
• Provides extra registers for RS/RegFile
– Is a source for operands
• Four fields/entry
– Instruction type

• Register number or store address
– Value field
• Holds value to write to register or data for store
– Ready field
• Has instruction finished executing?
• Note: store buffers from previous version now in ROB

Instruction Execution Sequence

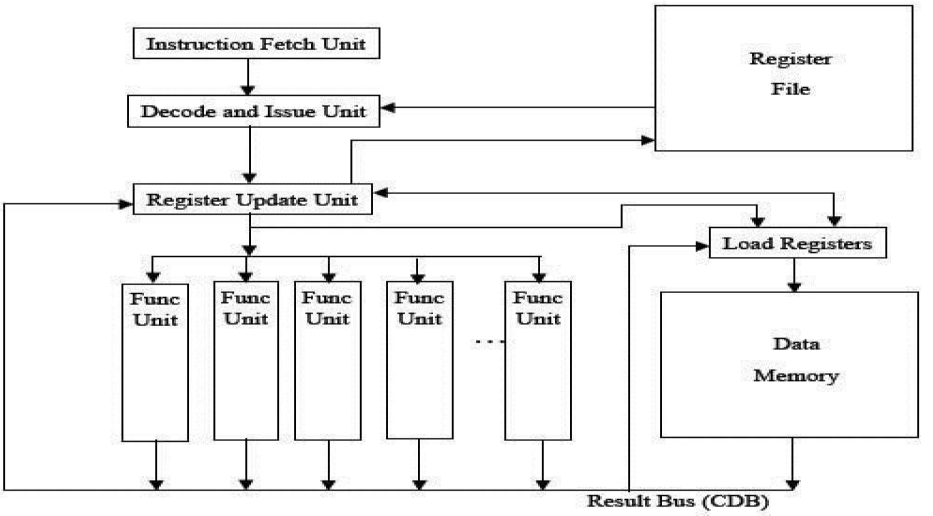
• Issue
– Issue instruction if opening in RS & ROB
– Send operands to RS from RegFile and/or ROB
• Execute
– Essentially the same as before
• Write Result
– Similar to before, but put result into ROB

• Commit (next slide)

Committing Instructions

Look at head of ROB
• Three types of instructions
– Incorrectly predicted branch
• Indicates speculation was wrong
• Flush ROB
• Execution restarts at proper location – Store
• Update memory
• Remove store from ROB
– Everything else
• Update registers
• Remove instruction from ROB

RUU Superscalar Computers



Modeling tool SimpleScalar implements an RUU style processor
– You will be using this tool after Spring Break
• Architecture similar to speculative Tomasulo's
• Register Update Unit (RUU)
– Controls instructions scheduling and dispatching to functional units
– Stores intermediate source values for instructions
– Ensures instruction commit occurs in order!
– Needs to be of appropriate size
• Minimum of issue width * number of pipeline stages
• Too small of an RUU can be a structural hazard!
• Result bus could be a structural hazard

3. What are the key issues in implementing advanced speculation techniques? Explain in detail? (June/July 2013)(June 2016)

1. separate execution from completion: instructions to execute speculatively but no *instructions update registers or memory until no more speculative*

2. therefore, add a final step – after an instruction is no longer speculative, called *instruction commit*– when it is allowed to make register and memory updates

3. allow instructions to execute and complete out of order but force them to commit in order

4. Add hardware called the *reorder buffer* (*ROB*), with registers to hold the result of an instruction *between completion and commit*

Tomasulo's Algorithm with Speculation: Four Stages

1. Issue: get instruction from Instruction Queue
   _ if reservation station and ROB slot free (no structural hazard),
   control issues instruction to reservation station and ROB, and sends to reservation station operand values (or reservation station source for values) as well as allocated ROB slot number

2. Execution: operate on operands (EX)
_ when both operands ready then execute;if not ready, watch CDB for result

3. Write result: finish execution (WB)
_ write on CDB to all awaiting units and ROB; mark reservation station available

4. Commit: update register or memory with ROB result
   _ when instruction reaches head of ROB and results present, update register with result or store to memory and remove instruction from ROB

   _ if an incorrectly predicted branch reaches the head of ROB, flush the ROB, and restart at correct successor of branch

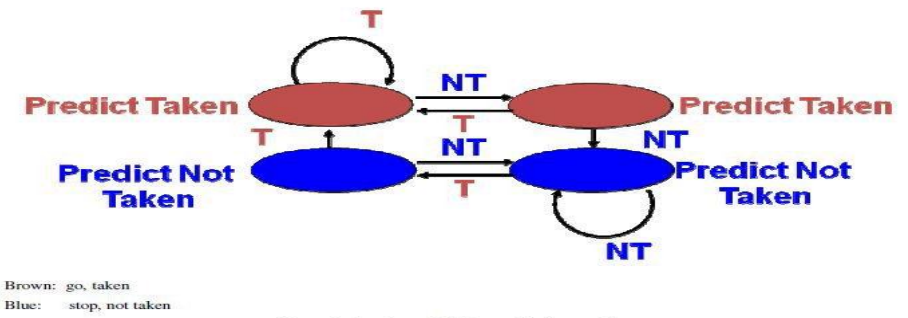
4 .Write a note on value predictors. (June/July 2014) (June 2016)

There are a small number of important branches in programs which have dynamic behavior for which dynamic branch prediction performance will be definitely better compared to static branch prediction.

• Performance = *f*(accuracy, cost of misprediction)
• Branch History Table (BHT) is used to dynamically predict the outcome of the current branch instruction. Lower bits of PC address index table of 1-bit values
– Says whether or not branch taken last time
- No address check

• Problem: in a loop, 1-bit BHT will cause two mispredictions (average is 9 iterations before exit):
– End of loop case, when it exits instead of looping as before
– First time through loop on *next* time through code, when it predicts exit instead of looping

• Simple two bit history table will give better performance. The four different states of 2 bit predictor is shown in the state transition diagram.



The states in a 2-bit prediction scheme

Correlating Branch Predictor

It may be possible to improve the prediction accuracy by considering the recent behavior of other branches rather than just the branch under consideration. Correlating predictors are two-level predictors. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch.

• Idea: record *m* most recently executed branches as taken or not taken, and use that pattern to select the proper *n*-bit branch history table (BHT)

• In general, (*m*,*n*) predictor means record last *m* branches to select between 2m history
tables, each with *n*-bit counters

– Thus, old 2-bit BHT is a (0,2) predictor
– Global Branch History: *m*-bit shift register keeping T/NT status of last *m*

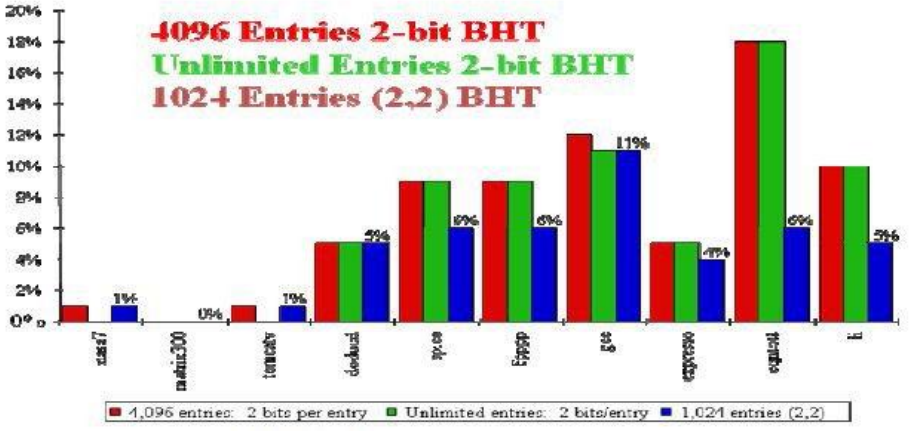
branches.

• Each entry in table has *m n*-bit predictors. In case of (2,2) predictor, behavior of recent

branches selects between four predictions of next branch, updating just that prediction.

The scheme of the table is shown:

Comparisons of different schemes are shown in the graph.



Comparison of 2 bit predictors (y-axis: % frequency of mispredictions, x-axis: SPEC89 programs)

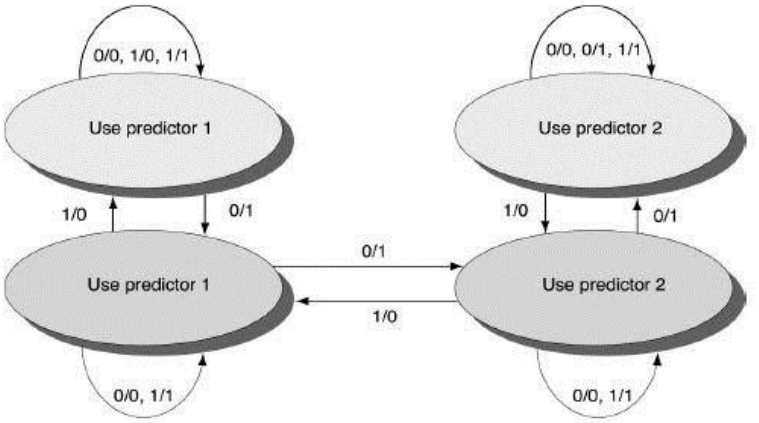
Tournament predictor is a multi level branch predictor and uses n bit saturating counter

to chose between predictors. The predictors used are global predictor and local predictor.

– Advantage of tournament predictor is ability to select the right predictor for a particular branch which is particularly crucial for integer benchmarks.

– A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks

– Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor was most effective oin recent prediction.



5. Explain branch-target buffer. (Dec 2013) (Jan 2015)(Jan 2016)

ROB Data Structure

ROB entry fields
• Instruction type: branch, store, register operation (i.e., ALU or load)
• State: indicates if instruction has completed and value is ready
• Destination: where result is to be written – register number for register operation (i.e. ALU or load), memory address for store
• branch has no destination result
Value: holds the value of instruction result till time to commit
Additional reservation station field
• Destination: Corresponding ROB entry number

Example
1. L.D F6, 34(R2)

2. L.D F2, 45(R3

3. MUL.D F0, F2, F4

4. SUB.D F8, F2, F6

5. DIV.D F10, F0, F6

6. ADD.D F6, F8, F2

The position of Reservation stations, ROB and FP registers are indicated below:

Assume latencies load 1 clock, add 2 clocks, multiply 10 clocks, divide 40 clocks
Show data structures just before MUL.D goes to commit*…*

Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	yes	MUL	Mem[45+Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV		Mem[34+Regs[R2]]	#3		#5	

At the time MUL.D is ready to commit only the two L.D instructions have already committed,though others have completed execution
Actually, the MUL.D is at the head of the ROB – the L.D instructions are shown only for understanding purposes #X represents value field of ROB entry number X

Floating point registers

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder#	3						6		4	5
Busy	yes	no	no	no	no	no	yes	...	yes	yes

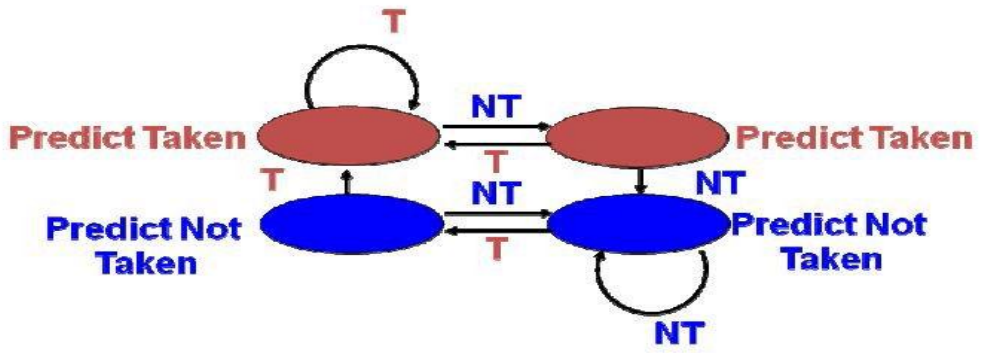
6. Write a short note on value predictor. (Jan 2014) (June 2015)(June 2016)

There are a small number of important branches in programs which have dynamic behavior for which dynamic branch prediction performance will be definitely better compared to static branch prediction.

- Performance = *f*(accuracy, cost of misprediction)
- Branch History Table (BHT) is used to dynamically predict the outcome of the current branch instruction. Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
    - No address check

• Problem: in a loop, 1-bit BHT will cause two mispredictions (average is 9 iterations before exit):
- End of loop case, when it exits instead of looping as before
- First time through loop on *next* time through code, when it predicts exit instead of looping

• Simple two bit history table will give better performance. The four different states of 2 bit predictor is shown in the state transition diagram.



Brown: go, taken
Blue: stop, not taken

The states in a 2-bit prediction scheme

Correlating Branch Predictor

It may be possible to improve the prediction accuracy by considering the recent behavior

of other branches rather than just the branch under consideration. Correlating predictors

are two-level predictors. Existing correlating predictors add information about the

behavior of the most recent branches to decide how to predict a given branch.

• Idea: record *m* most recently executed branches as taken or not taken, and use that

pattern to select the proper *n*-bit branch history table (BHT)

• In general, (*m*,*n*) predictor means record last *m* branches to select between 2m history

tables, each with *n*-bit counters

– Thus, old 2-bit BHT is a (0,2) predictor

– Global Branch History: *m*-bit shift register keeping T/NT status of last *m*

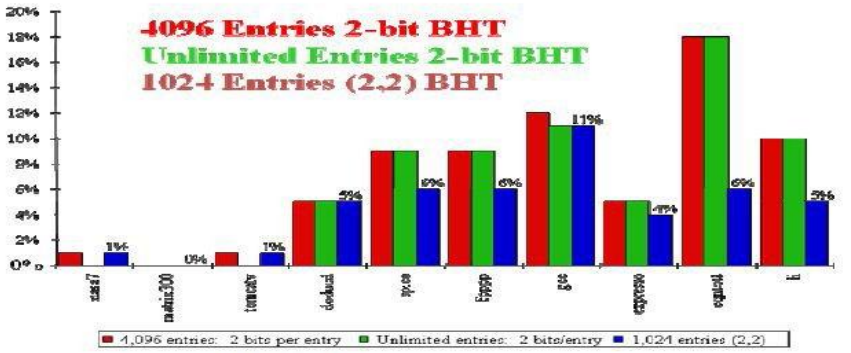
branches.

• Each entry in table has *m* *n*-bit predictors. In case of (2,2) predictor, behavior of recent

branches selects between four predictions of next branch, updating just that prediction.

The scheme of the table is shown:

Comparisons of different schemes are shown in the graph.



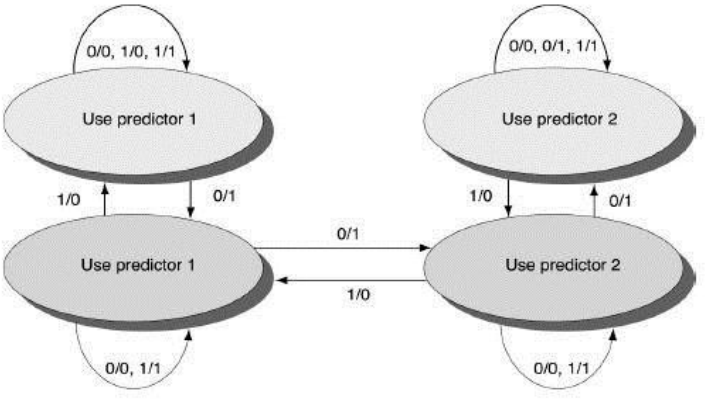
Comparison of 2 bit predictors (y-axis: % frequency of mispredictions, x-axis: SPEC89 programs)

Tournament predictor is a multi level branch predictor and uses n bit saturating counter to chose between predictors. The predictors used are global predictor and local predictor.

– Advantage of tournament predictor is ability to select the right predictor for a particular branch which is particularly crucial for integer benchmarks.

– A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks

– Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor was most effective oin recent prediction.



7. what are the key issues in implementing advanced speculation techniques? Explain them in detail? (June2013) (Jan 2015)

Tomasulo's w/HW Speculation

• Key aspects of this design
– Separate forwarding (result bypassing) from actual instruction completion
• Assuming instructions are executing speculatively
• Can pass results to later instructions, but prevents instruction from performing updates that can't be "undone"
– Once instruction is no longer speculative it can update register file/memory
• New step in execution sequence: instruction commit
• Requires instructions to wait until they can commit Commits still happen in order
Reorder Buffer (ROB)
Instructions hang out here before committing
• Provides extra registers for RS/RegFile
– Is a source for operands
• Four fields/entry
– Instruction type
• Branch, store, or register operation (ALU & load)
– Destination field
• Register number or store address
– Value field
• Holds value to write to register or data for store
– Ready field
• Has instruction finished executing?
• Note: store buffers from previous version now in ROB

Instruction Execution Sequence

• Issue
– Issue instruction if opening in RS & ROB
– Send operands to RS from RegFile and/or ROB
• Execute
– Essentially the same as before
• Write Result
– Similar to before, but put result into ROB
• Commit (next slide)

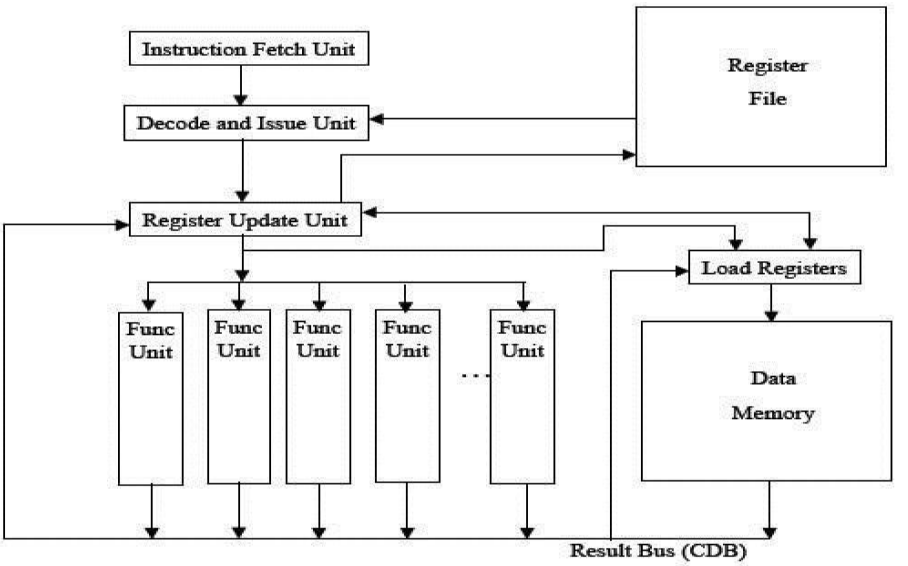
Committing Instructions

Look at head of ROB
• Three types of instructions
– Incorrectly predicted branch
• Indicates speculation was wrong

• Flush ROB
• Execution restarts at proper location – Store
• Update memory

• Remove store from ROB
– Everything else
• Update registers
• Remove instruction from ROB

RUU Superscalar Computers



Modeling tool SimpleScalar implements an RUU style processor
– You will be using this tool after Spring Break
• Architecture similar to speculative Tomasulo's
• Register Update Unit (RUU)
– Controls instructions scheduling and dispatching to functional units
– Stores intermediate source values for instructions
– Ensures instruction commit occurs in order!
– Needs to be of appropriate size
• Minimum of issue width * number of pipeline stages
• Too small of an RUU can be a structural hazard!
• Result bus could be a structural hazard

PART B

UNIT 5

1. Explain the basic schemes for enforcing coherence in a shared memory multiprocessor system? (Jun 2014) (June2013) (Jan 2015)(June 2016)

Cache Coherence

Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two different values. I.e, If two different processors have two different values for the same location, this difficulty is generally referred to as cache coherence problem

Time	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

Cache coherence problem for a single memory location

• **Informally:**

– "Any read must return the most recent write"
– Too strict and too difficult to implement
–

• **Better:**

– "Any write must eventually be seen by a read"
– All writes are seen in proper order ("serialization")
–

• **Two rules to ensure this:**

– "If P writes x and then P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"
– Writes to a single location are serialized: seen in one order
  • Latest write will be seen
  • Otherwise could see writes in illogical order (could see older value after a newer value)

The definition contains two different aspects of memory system:

- Coherence
- Consistency

A memory system is coherent if,

- Program order is preserved.
- Processor should not continuously read the old data value.
- Write to the same location are serialized.

The above three properties are sufficient to ensure coherence,*When a written value will be seen is also important.* This issue is defined by memory consistency model. Coherence and consistency are complementary.

Basic schemes for enforcing coherence

Coherence cache provides:

- migration: a data item can be moved to a local cache and used there in a transparent fashion.
- replication for shared data that are being simultaneously read.
- both are critical to performance in accessing shared data.

To over come these problems, adopt a hardware solution by introducing a protocol tomaintain coherent caches named as Cache Coherence Protocols
These protocols are implemented for tracking the state of any sharing of a data block.
Two classes of Protocols

- Directory based
- Snooping based

Directory based

- Sharing status of a block of physical memory is kept in one location called the directory.
- Directory-based coherence has slightly higher implementation overhead than snooping.
- It can scale to larger processor count.

Snooping

- Every cache that has a copy of data also has a copy of the sharing status of the block.
- No centralized state is kept.
- Caches are also accessible via some broadcast medium (bus or switch)
- Cache controller monitor or snoop on the medium to determine whether or not they have a copy of a block that is represented on a bus or switch access.

Snooping protocols are popular with multiprocessor and caches attached to single shared memory as they can use the existing physical connection- bus to memory, to interrogate the status of the caches. Snoop based cache coherence scheme is implemented on a shared bus. Any communication medium that broadcasts cache misses to all the

processors.

Basic Snoopy Protocols

- Write strategies
  - – Write-through: memory is always up-to-date
  - – Write-back: snoop in caches to find most recent copy
- Write Invalidate Protocol
  - – Multiple readers, single writer
  - – Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies
- Read miss: further read will miss in the cache and fetch a new copy of the data.
- Write Broadcast/Update Protocol (typically write through)
  - – Write to shared data: broadcast on bus, processors snoop, and *update* any copies
  - – Read miss: memory/cache is always up-to-date.
- Write serialization: bus serializes requests!
- – Bus is single point of arbitration

Examples of Basic Snooping Protocols

Write Invalidate

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.

Write Update

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

An example of a write update or broadcast protocol working on a snooping bus for a single cache block (X) with write-back caches.

Assume neither cache initially holds X and the value of X in memory is 0

Example Protocol

• Snooping coherence protocol is usually implemented by incorporating a finitestate controller in each node
• Logically, think of a separate controller associated with each cache block
  – That is, snooping operations or cache requests for different blocks can proceed independently
• In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
  – that is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time

Example Write Back Snoopy Protocol

• Invalidation protocol, write-back cache
  – Snoops every address on bus
  – If it has a dirty copy of requested block, provides that block in response to the read request and aborts the memory access
• Each memory block is in one state:
  – Clean in all caches and up-to-date in memory (Shared)
  – OR Dirty in exactly one cache (Exclusive)
  – OR Not in any caches
• Each cache block is in one state (track these):
  – Shared : block can be read
  – OR Exclusive : cache has only copy, its writeable, and dirty
  – OR Invalid : block contains no data (in uniprocessor cache too)
• Read misses: cause all caches to snoop bus
• Writes to clean blocks are treated as misses

2. Expalin the directory based coherence for a distributed memory multiprocessor system? (Jun 2014) (June 2015)(Jan 2016)

Directory Protocols

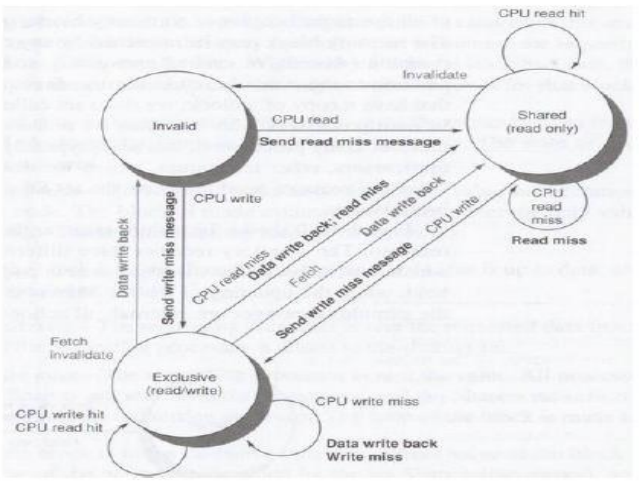
• Similar to Snoopy Protocol: Three states
– ***Shared***: 1 or more processors have the block cached, and the value in memory is up-to-date (as well as in all the caches)
– ***Uncached***: no processor has a copy of the cache block (not valid in any cache)
– ***Exclusive***: Exactly one processor has a copy of the cache block, and it has written the block, so the memory copy is out of date
• The processor is called the owner of the block
• In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared (usually a bit vector for each memory block: 1 if processor has copy)
• Keep it simple(r):
– Writes to non-exclusive data => write miss
– Processor blocks until access completes
– Assume messages received and acted upon in order sent

Messages for Directory Protocols

Message type	Source	Destination	Message contents	Function of this message
Read miss	local cache	home directory	P, A	Processor P has a read miss at address A; request data and make P a read sharer.
Write miss	local cache	home directory	P, A	Processor P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	home directory	remote cache	A	Invalidate a shared copy of data at address A.
Fetch	home directory	remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	home directory	remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	home directory	local cache	D	Return a data value from the home memory.
Data write back	remote cache	home directory	A, D	Write back a data value for address A.

• local node: the node where a request originates
• home node: the node where the memory location and directory entry of an address reside
• remote node: the node that has a copy of a cache block (exclusive or shared)

State Transition Diagram for Individual Cache Block



• Comparing to snooping protocols:
  – identical states
  – stimulus is almost identical
  – write a shared cache block is treated as a write miss (without fetch the block)
  – cache block must be in exclusive state when it is written
  – any shared block must be up to date in memory

• write miss: data fetch and selective invalidate operations sent by the directory controller (broadcast in snooping protocols)

Directory Operations: Requests and Actions

• Message sent to directory causes two actions:
  – Update the directory
  – More messages to satisfy request

• Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  – Read miss: requesting processor sent data from memory &requestor made only sharing node; state of block made Shared.
  – Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

• Block is Shared => the memory value is up-to-date:
  – Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  – Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

• Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:
  – Read miss: owner processor sent data fetch message, causing state of

block in owner's cache to transition to Shared and causes owner to send data to

directory, where it is written to memory & sent back to requesting processor.

Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.

– Data write-back: owner processor is replacing the block and hence must write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.

– Write miss: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

3. Explain the directory based cache coherence for a distributed memory multiprocessor system along with state transition diagram. (June/July 2013) (Jan 2014)

Cache Coherence

Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two different values. I.e, If two different processors have two different values for the same location, this difficulty is generally referred to as cache coherence problem

Time	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

Cache coherence problem for a single memory location

• **Informally:**

– "Any read must return the most recent write"

– Too strict and too difficult to implement

–

• **Better:**

– "Any write must eventually be seen by a read"

– All writes are seen in proper order ("serialization")

–

• **Two rules to ensure this:**

– "If P writes x and then P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"

– Writes to a single location are serialized: seen in one order
• Latest write will be seen
• Otherwise could see writes in illogical order (could see older value after a newer value)

The definition contains two different aspects of memory system:
• Coherence
• Consistency

A memory system is coherent if,
• Program order is preserved.
• Processor should not continuously read the old data value.
• Write to the same location are serialized.

The above three properties are sufficient to ensure coherence,*When a written value will be seen is also important.* This issue is defined by memory consistency model. Coherence and consistency are complementary.

Basic schemes for enforcing coherence

Coherence cache provides:
• migration: a data item can be moved to a local cache and used there in a transparent fashion.
• replication for shared data that are being simultaneously read.
• both are critical to performance in accessing shared data.

To over come these problems, adopt a hardware solution by introducing a protocol to maintain coherent caches named as Cache Coherence Protocols

These protocols are implemented for tracking the state of any sharing of a data block.

Two classes of Protocols
• Directory based
• Snooping based

Directory based
• Sharing status of a block of physical memory is kept in one location called the directory.
• Directory-based coherence has slightly higher implementation overhead than snooping.
• It can scale to larger processor count.

Snooping
• Every cache that has a copy of data also has a copy of the sharing status of the block.
• No centralized state is kept.
• Caches are also accessible via some broadcast medium (bus or switch)

• Cache controller monitor or snoop on the medium to determine whether or not they have a copy of a block that is represented on a bus or switch access.

Snooping protocols are popular with multiprocessor and caches attached to single shared memory as they can use the existing physical connection- bus to memory, to interrogate the status of the caches. Snoop based cache coherence scheme is implemented on a shared bus. Any communication medium that broadcasts cache misses to all the processors

Basic Snoopy Protocols

• Write strategies

– Write-through: memory is always up-to-date

– Write-back: snoop in caches to find most recent copy

• Write Invalidate Protocol

– Multiple readers, single writer

– Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies

• Read miss: further read will miss in the cache and fetch a new copy of the data.

• Write Broadcast/Update Protocol (typically write through)

– Write to shared data: broadcast on bus, processors snoop, and *update* any copies

– Read miss: memory/cache is always up-to-date.

• Write serialization: bus serializes requests!

– Bus is single point of arbitration

Examples of Basic Snooping Protocols

Write Invalidate

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.

Write Update

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

An example of a write update or broadcast protocol working on a snooping bus for a single cache block (X) with write-back caches.

Example Protocol

• Snooping coherence protocol is usually implemented by incorporating a finitestate controller in each node

• Logically, think of a separate controller associated with each cache block
– That is, snooping operations or cache requests for different blocks can proceed independently
• In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
– that is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time

Example Write Back Snoopy Protocol

• Invalidation protocol, write-back cache

– Snoops every address on bus
– If it has a dirty copy of requested block, provides that block in response to the read request and aborts the memory access
• Each memory block is in one state:
– Clean in all caches and up-to-date in memory (Shared)
– OR Dirty in exactly one cache (Exclusive)
– OR Not in any caches
• Each cache block is in one state (track these):
– Shared : block can be read
– OR Exclusive : cache has only copy, its writeable, and dirty
– OR Invalid : block contains no data (in uniprocessor cache too)
• Read misses: cause all caches to snoop bus

• Writes to clean blocks are treated as misses

4. Explain any two hardware primitive to implement synchronization with example. (June/July 2014) (Dec 2013)(June 2016)

Synchronization: The Basics

Synchronization mechanisms are typically built with user-level software routines that rely on hardware –supplied synchronization instructions.

• Why Synchronize?

Need to know when it is safe for different processes to use shared data

• Issues for Synchronization:

– Uninterruptable instruction to fetch and update memory (atomic operation);

– User level synchronization operation using this primitive;

– For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Uninterruptable Instruction to Fetch and Update Memory

• Atomic exchange: interchange a value in a register for a value in memory

0 _ synchronization variable is free

1 _ synchronization variable is locked and unavailable

– Set register to 1 & swap

– New value in register determines success in getting lock

0 if you succeeded in setting the lock (you were first)

1 if other processor had already claimed access

– Key is that exchange operation is indivisible

• Test-and-set: tests a value and sets it if the value passes the test

• Fetch-and-increment: it returns the value of a memory location and atomically increments it

– 0 _ synchronization variable is free

• Hard to have read & write in 1 instruction: use 2 instead

• Load linked (or load locked) + store conditional

– Load linked returns the initial value

– Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise

• Example doing atomic swap with LL & SC:

```
try:    mov     R3,R4 ;         mov exchange value

        ll      R2,0(R1) ; load linked

        sc      R3,0(R1) ; store conditional

        beqz    R3,try ; branch store fails (R3 = 0)
```

```
        mov   R4,R2 ; put load value in R4
```

• Example doing fetch & increment with LL & SC:

```
try:    ll      R2,0(R1) ;    load linked
        addi    R2,R2,#1 ;    increment (OK if reg–reg)
        sc      R2,0(R1) ;    store conditional
        beqz    R2,try ;      branch store fails (R2 = 0)
```

User Level Synchronization—Operation Using this Primitive

• Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
li  R2,#1
lockit: exch    R2,0(R1) ;    atomic exchange
        bnez    R2,lockit ;   already locked?
```

• What about MP with cache coherency?

– Want to spin on cache copy to avoid full memory latency

– Likely to get cache hits for such variables

• Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

• Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):

```
try:    li      R2,#1
lockit: lw      R3,0(R1)      ;load var
        bnez    R3,lockit ;   _ 0 _ not free _ spin
        exch    R2,0(R1) ;    atomic exchange
        bnez    R2,try ;      already locked?
```

Memory Consistency Models

• What is consistency? When must a processor see the new value? e.g., seems that P1: A = 0; P2: B = 0;

```
..... .....
        A = 1; B = 1;
   L1: if (B == 0) ... L2: if (A == 0) ...
```

• Impossible for both if statements L1 & L2 to be true?

– What if write invalidate is delayed & processor continues?

• Memory consistency models:

what are the rules for such cases?

• Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved _ assignments before ifs above

– SC: delay all memory accesses until all invalidates done

• Schemes faster execution to sequential consistency

• Not an issue for most programs; they are synchronized

– A program is synchronized if all access to shared data are ordered by synchronization operations

write (x)

...
release (s) *{unlock}*
...
acquire (s) *{lock}*
...
read(x)

• Only those programs willing to be nondeterministic are not synchronized: "data race": outcome f(proc. speed)

• Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses

Relaxed Consistency Models : The Basics

• Key idea: allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent

– By relaxing orderings, may obtain performance advantages

– Also specifies range of legal compiler optimizations on shared data

– Unless synchronization points are clearly defined and programs are synchronized, compiler could not interchange read and write of 2 shared data items because might affect the semantics of the program

• 3 major sets of relaxed orderings:

1. W_R ordering (all writes completed before next read)

• Because retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization. Called processor consistency

2. W _ W ordering (all writes completed before next write)
3. R _ W and R _ R orderings, a variety of models depending on ordering restrictions and how synchronization operations enforce ordering

• Many complexities in relaxed consistency models; defining precisely what it means for a write to complete; deciding when processors can see values that it has written

5. List and explain any three hardware primitives to implement synchronization. (June2013) (June 2015)

Synchronization: The Basics

Synchronization mechanisms are typically built with user-level software routines that rely on hardware –supplied synchronization instructions.

• Why Synchronize?

Need to know when it is safe for different processes to use shared data

• Issues for Synchronization:

– Uninterruptable instruction to fetch and update memory (atomic operation);

– User level synchronization operation using this primitive;

– For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Uninterruptable Instruction to Fetch and Update Memory

• Atomic exchange: interchange a value in a register for a value in memory

0 _ synchronization variable is free

1 _ synchronization variable is locked and unavailable

– Set register to 1 & swap

– New value in register determines success in getting lock

0 if you succeeded in setting the lock (you were first)

1 if other processor had already claimed access

– Key is that exchange operation is indivisible

• Test-and-set: tests a value and sets it if the value passes the test

• Fetch-and-increment: it returns the value of a memory location and atomically increments it

– 0 _ synchronization variable is free

• Hard to have read & write in 1 instruction: use 2 instead

• Load linked (or load locked) + store conditional

– Load linked returns the initial value

– Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise

• Example doing atomic swap with LL & SC:

```
try:    mov    R3,R4 ;        mov exchange value

        ll     R2,0(R1) ; load linked

        sc     R3,0(R1) ; store conditional

        beqz   R3,try ; branch store fails (R3 = 0)

        mov    R4,R2 ; put load value in R4
```

• Example doing fetch & increment with LL & SC:

```
try:    ll     R2,0(R1) ;     load linked
        addi   R2,R2,#1 ;      increment (OK if reg–reg)
        sc     R2,0(R1) ;     store conditional
        beqz   R2,try ;       branch store fails (R2 = 0)
```

User Level Synchronization—Operation Using this Primitive

• Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
li  R2,#1
lockit: exch   R2,0(R1) ;     atomic exchange
        bnez   R2,lockit ;    already locked?
```

• What about MP with cache coherency?

– Want to spin on cache copy to avoid full memory latency

– Likely to get cache hits for such variables

• Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

• Solution: start by simply repeatedly reading the variable; when it changes, then

```
try      exchange      ("test and test&set"):
try:     li      R2,#1
lockit:  lw      R3,0(R1)      ;load var
bnez     R3,lockit ;           _ 0 _ not free _ spin
exch     R2,0(R1) ;            atomic exchange
bnez     R2,try ;              already locked?
```

Memory Consistency Models

• What is consistency? When must a processor see the new value? e.g., seems that P1: A = 0; P2: B = 0;

```
..... .....
        A = 1; B = 1;
   L1: if (B == 0) ... L2: if (A == 0) ...
```

• Impossible for both if statements L1 & L2 to be true?

– What if write invalidate is delayed & processor continues?

• Memory consistency models:

what are the rules for such cases?

• Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved _ assignments before ifs above

– SC: delay all memory accesses until all invalidates done

• Schemes faster execution to sequential consistency

• Not an issue for most programs; they are synchronized

– A program is synchronized if all access to shared data are ordered by synchronization operations

```
write (x)
...

release (s) {unlock}
...
acquire (s) {lock}
...
read(x)
```

• Only those programs willing to be nondeterministic are not synchronized: "data race": outcome f(proc. speed)

• Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses

Relaxed Consistency Models : The Basics

• Key idea: allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves

as if the processor were sequentially consistent

– By relaxing orderings, may obtain performance advantages

– Also specifies range of legal compiler optimizations on shared data

– Unless synchronization points are clearly defined and programs are synchronized, compiler could not interchange read and write of 2 shared data items because might affect the semantics of the program

• 3 major sets of relaxed orderings:

1. W_R ordering (all writes completed before next read)

• Because retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization. Called processor consistency

2. W _ W ordering (all writes completed before next write)

3. R _ W and R _ R orderings, a variety of models depending on ordering restrictions and how synchronization operations enforce ordering

• Many complexities in relaxed consistency models; defining precisely what it means for a write to complete; deciding when processors can see values that it has written

6. Explain the directory based coherence for a distributed memory multiprocessor system? (Jan 2014) (Jan 2015)(Jan 2016)

Directory Protocols

• Similar to Snoopy Protocol: Three states

– ***Shared***: 1 or more processors have the block cached, and the value in memory is up-to-date (as well as in all the caches)

– ***Uncached***: no processor has a copy of the cache block (not valid in any cache)

– ***Exclusive***: Exactly one processor has a copy of the cache block, and it has written the block, so the memory copy is out of date

• The processor is called the owner of the block

• In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared (usually a bit vector for each memory block: 1 if processor has copy)

• Keep it simple(r):

– Writes to non-exclusive data => write miss

– Processor blocks until access completes

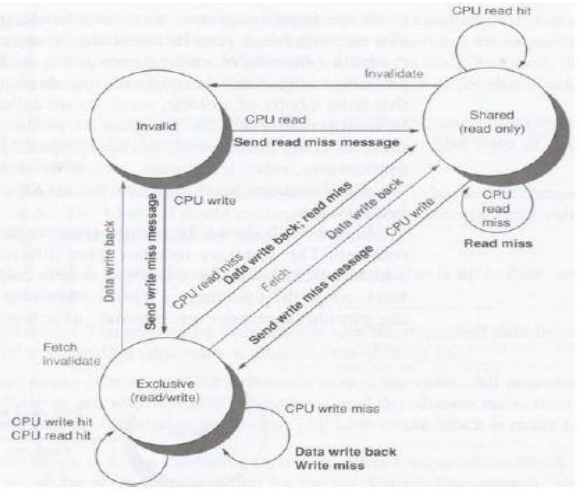
– Assume messages received and acted upon in order sent

local node: the node where a request originates

• home node: the node where the memory location and directory entry of an address reside

• remote node: the node that has a copy of a cache block (exclusive or shared)

State Transition Diagram for Individual Cache Block



• Comparing to snooping protocols:

– identical states

– stimulus is almost identical

– write a shared cache block is treated as a write miss (without fetch the block)

– cache block must be in exclusive state when it is written

– any shared block must be up to date in memory

• write miss: data fetch and selective invalidate operations sent by the directory controller (broadcast in snooping protocols)

Directory Operations: Requests and Actions

• Message sent to directory causes two actions:

– Update the directory

– More messages to satisfy request

• Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:

– Read miss: requesting processor sent data from memory &requestor made only sharing node; state of block made Shared.

– Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

• Block is Shared => the memory value is up-to-date:

– Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.

– Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

• Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:

– Read miss: owner processor sent data fetch message, causing state of block in owner's cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor.

Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.

– Data write-back: owner processor is replacing the block and hence must

write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.

– Write miss: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

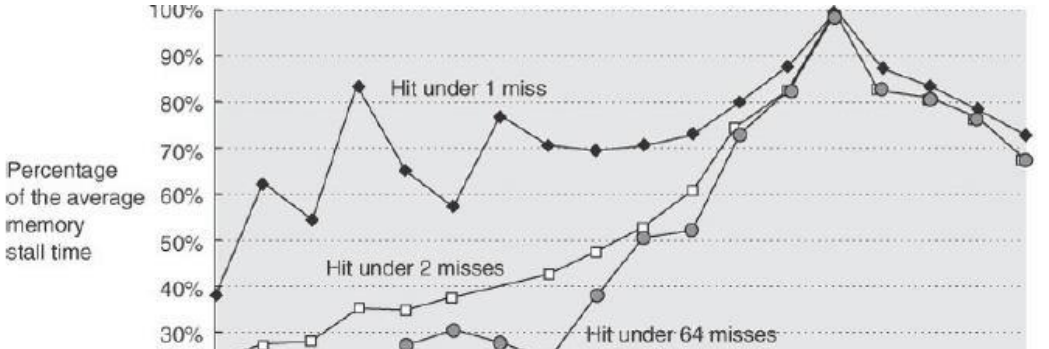
UNIT 6

1. Assume we have a computer where the clock per instruction(CPI) is 1.0 when all memory accesses hit the cache. the only data accesses are loads and stores and these total 50 % of the instructions. if the mass penality is 25 clock cycles and the mass rate is 2%.how much faster would the computer be if all instructions were cache hits? (June 2014) (June2013)(Jan 2014)

• FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26

• Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19

• 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss, SPEC 92



2. Explain in brief ,the types of basic cache optimization? (June 2014) (June2013)(Jan 2016)

Cache Optimizations

Six basic cache optimizations

1. Larger block size to reduce miss rate:

- To reduce miss rate through spatial locality.
- Increase block size.
- Larger block size reduce compulsory misses.
- But they increase the miss penalty.

2. Bigger caches to reduce miss rate:

- capacity misses can be reduced by increasing the cache capacity.
- Increases larger hit time for larger cache memory and higher cost and power.

3. Higher associativity to reduce miss rate:

- Increase in associativity reduces conflict misses.

4. Multilevel caches to reduce penalty:

- Introduces additional level cache
- Between original cache and memory.
- L1- original cache

L2- added cache.

L1 cache: - small enough

- speed matches with clock cycle time.

L2 cache: - large enough

- capture many access that would go to main memory.

Average access time can be redefined as

Hit timeL1+ Miss rate L1 X (Hit time L2 + Miss rate L2 X Miss penalty L2)

5. Giving priority to read misses over writes to reduce miss penalty:

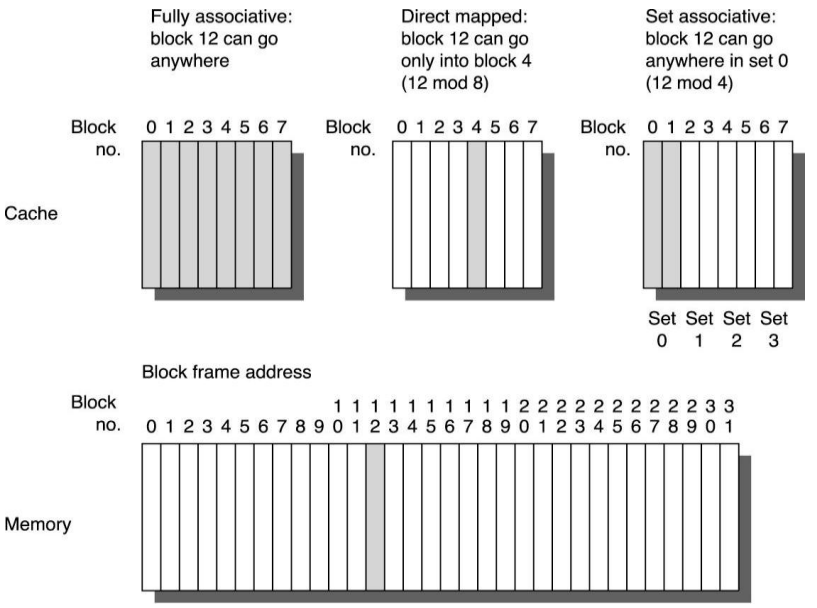
- write buffer is a good place to implement this optimization.
- write buffer creates hazards: read after write hazard.

6. Avoiding address translation during indexing of the cache to reduce hit time:

- Caches must cope with the translation of a virtual address from the processor to a physical address to access memory.
- common optimization is to use the page offset.

- part that is identical in both virtual and physical addresses- to index the cache.

3. Explain block replacement strategies to replace a block, with example when a cache. (Jan 2015)(June 2016)



4. Explain the types of basic cache optimization. (Jan 2014) (Jan 2015)

Cache Optimizations

Six basic cache optimizations

1. Larger block size to reduce miss rate:

- To reduce miss rate through spatial locality.
- Increase block size.
- Larger block size reduce compulsory misses.
- But they increase the miss penalty.

2. Bigger caches to reduce miss rate:

- capacity misses can be reduced by increasing the cache capacity. Increases larger hit time for larger cache memory and higher cost and power.

3. Higher associativity to reduce miss rate:

- Increase in associativity reduces conflict misses.

4. Multilevel caches to reduce penalty:

- Introduces additional level cache
- Between original cache and memory.
- L1- original cache

L2- added cache.

L1 cache: - small enough

- speed matches with clock cycle time.

L2 cache: - large enough

- capture many access that would go to main memory.

Average access time can be redefined as

Hit timeL1+ Miss rate L1 X (Hit time L2 + Miss rate L2 X Miss penalty L2)

5. Giving priority to read misses over writes to reduce miss penalty:

- write buffer is a good place to implement this optimization.
- write buffer creates hazards: read after write hazard.

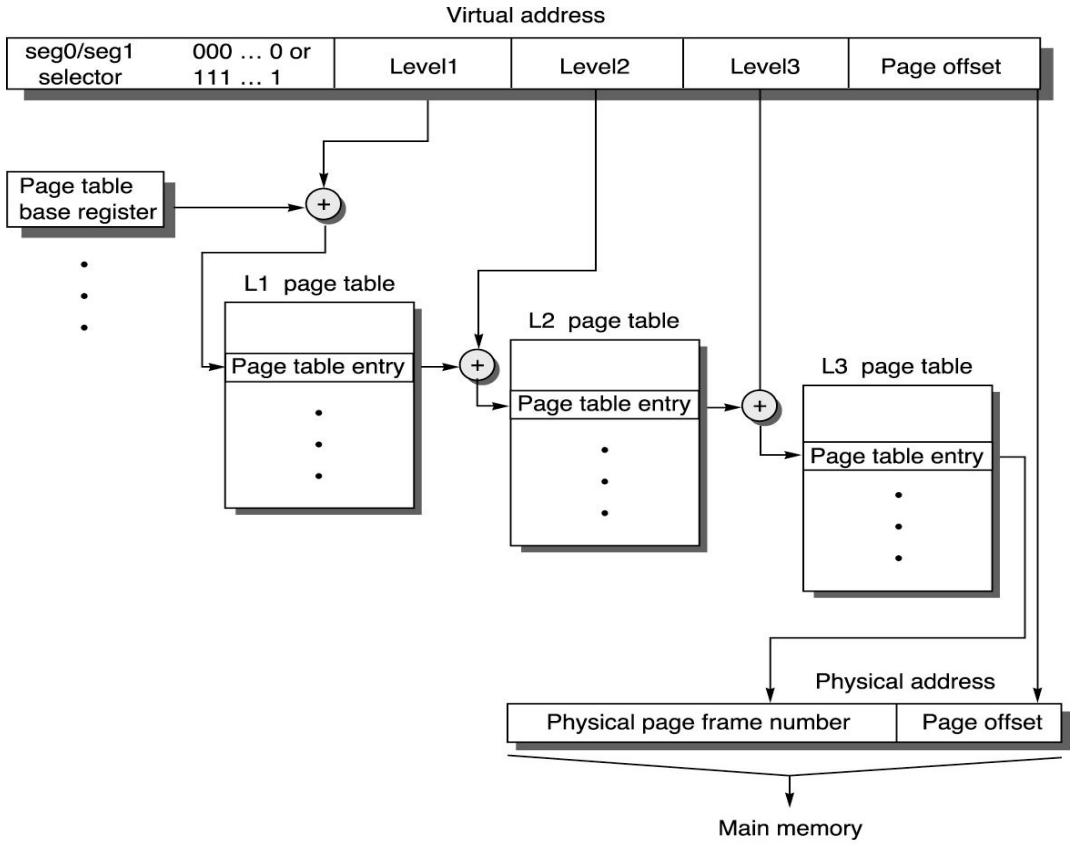
6. Avoiding address translation during indexing of the cache to reduce hit time:

- Caches must cope with the translation of a virtual address from the processor to

a physical address to access memory.
- common optimization is to use the page offset.

- part that is identical in both virtual and physical addresses- to index the cache.

5. With a diagram, explain organization of data cache in the opteron microprocessor. (June/July 2013)(Jan 2016)

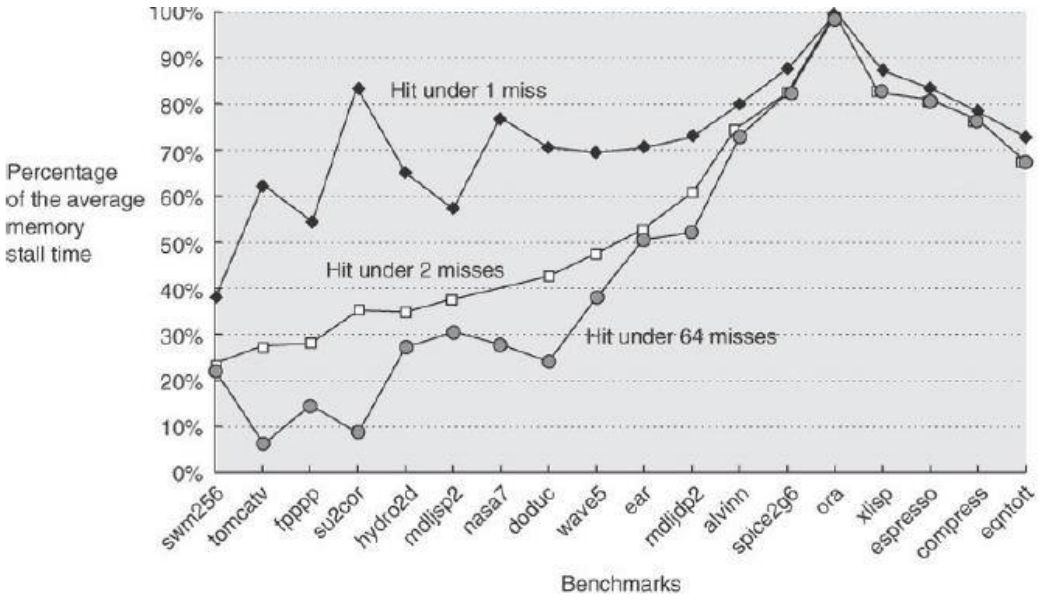


6. Assume we have a computer where CPI is 1.0 when all memory accesses hits in the cache. The only data accesses are loads and stores , and these 50% of the instruction. If the miss penalty is of 25 cycles and miss rate is 2% , how much faster the computer be , if all the instruction were cache hits ? (Jan 2014) (June-13) (July 2015)

• FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26

• Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19

• 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss, SPEC 92



7 . Briefly explain four basic cache optimization methods. (Dec2013) (June2013) (July 2015)(June 2016)

Six basic cache optimizations

1. Larger block size to reduce miss rate:

- To reduce miss rate through spatial locality.
- Increase block size.
- Larger block size reduce compulsory misses.
- But they increase the miss penalty.

2. Bigger caches to reduce miss rate:

- capacity misses can be reduced by increasing the cache capacity.
- Increases larger hit time for larger cache memory and higher cost and power.

3. Higher associativity to reduce miss rate:

- Increase in associativity reduces conflict misses.

4. Multilevel caches to reduce penalty:

- Introduces additional level cache
- Between original cache and memory.
- L1- original cache

L2- added cache.

L1 cache: - small enough

- speed matches with clock cycle time.

L2 cache: - large enough

- capture many access that would go to main memory.

Average access time can be redefined as

Hit timeL1+ Miss rate L1 X (Hit time L2 + Miss rate L2 X Miss penalty L2)

5. Giving priority to read misses over writes to reduce miss penalty:

- write buffer is a good place to implement this optimization.
- write buffer creates hazards: read after write hazard.

6. Avoiding address translation during indexing of the cache to reduce hit time:

- Caches must cope with the translation of a virtual address from the processor to a physical address to access memory.
- common optimization is to use the page offset.
- part that is identical in both virtual and physical addresses- to index the cache.

UNIT 7

1. Which are the major categories of the advanced optimization of cache perform ance? explain any one in details. (Jun 2014) (J a n 2 0 1 5) (June 2015)(Jan 2016)

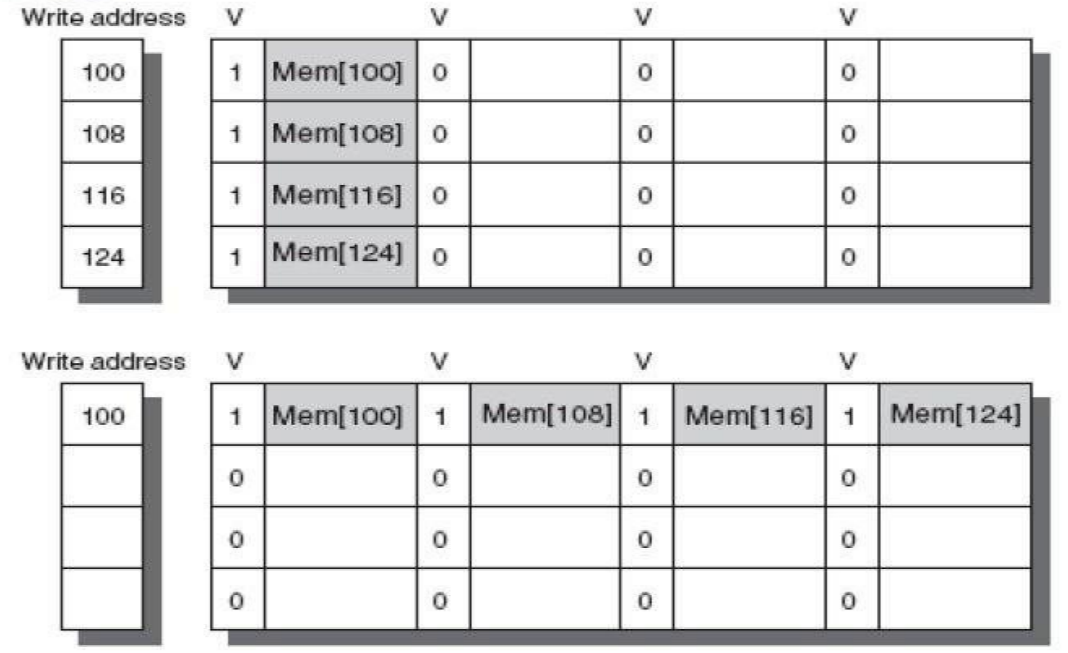
Advanced Cache Optimizations

- **Reducing hit time**
  1. Small and simple caches
  2. Way prediction
  3. Trace caches
- **Increasing cache bandwidth**
  4. Pipelined caches
  5. Multibanked caches
  6. Nonblocking caches
- **Reducing Miss Penalty**
  7. Critical word first
  8. Merging write buffers
- **Reducing Miss Rate**
  9.Compiler optimizations
- **Reducing miss penalty or miss rate via parallelism**
  10.Hardware prefetching
  11.Compiler prefetching

Merging Write Buffer to Reduce Miss Penalty

- Write buffer to allow processor to continue while waiting to write to memory
- If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry -If so, new data are combined with that entry
- Increases block size of write for write-through cache of writes to sequential words, bytes since multiword writes more efficient to memory
- The Sun T1 (Niagara) processor, among many others, uses write merging

To illustrate write merging



2.Explain in detail the architecture support for protecting processes from each other via virtual memory (Jun 2014) (June/July 2013)(Jan 2015)(June 2016)

Virtual Memory and Virtual Machines

Slide Sources: Based on "Computer Architecture" by Hennessy/Patterson.
Supplemented from various freely downloadable sources
Security and Privacy

- •Innovations in Computer Architecture and System software
- •Protection through Virtual Memory
- •Protection from Virtual Machines
  - –Architectural requirements
  - –Performance

Protection via Virtual Memory

- •Processes
  - –Running program
  - –Environment (state) needed to continue running it
- •Protect Processes from each other
  - –Page based virtual memory including TLB which caches page table entries –Example: Segmentation and paging in 80x86

Processes share hardware without interfering with each other

- Provide User Process and Kernel Process
- Readable portion of Processor state:
  - –User supervisor mode bit
  - –Exception enable/disable bit
  - –Memory protection information
- System call to transfer to supervisor mode
  - –Return like normal subroutine to user mode
- Mechanism to limit memory access

Memory protection

- Virtual Memory

–Restriction on each page entry in page table
–Read, write, execute privileges
–Only OS can update page table
–TLB entries also have protection field

- Bugs in OS

–Lead to compromising security
–Bugs likely due to huge size of OS code

Protection via Virtual Machines

Virtualization

- Goal:

– Run multiple instances of different OS on the same hardware
– Present a transparent view of one or more environments (M-to-N mapping of M "real" resources, N "virtual" resources)

Protection via Virtual Machines

Virtualization- cont.

- Challenges:

– Have to split all resources (processor, memory, hard drive, graphics card, networking card etc.) among the different OS -> virtualize the resources
  - The OS can not be aware that it is using virtual resources instead of real resources

Problems with virtualization

- Two components when using virtualization:

– Virtual Machine Monitor (VMM)
– Virtual Machine(s) (VM)

- Para-virtualization:

– Operating System has been modified in order to run as a VM

- 'Fully' Virtualized:
  - No modification required of an OS to run as a VM

Virtual Machine Monitor-'hypervisor'

- Isolates the state of each guest OS from each other
- Protects itself from guest software
- Determines how to map virtual resources to physical resources

– Access to privileged state
– Address translation

– I/O
– Exceptions and interrupts
• Relatively small code (compared to an OS)
• VMM must run in a higher privilege mode than guest OS

Managing Virtual Memory

• Virtual memory offers many of the features required for hardware virtualization
– Separates the physical memory onto multiple processes
– Each process 'thinks' it has a linear address space of full size
– Processor holds a page table translating virtual addresses used by a process and the according physical memory
– Additional information restricts processes from
• Reading a page of on another process or
• Allow reading but not modifying a memory page or
• Do not allow to interpret data in the memory page as instructions and do not try to execute them
• Virtual Memory management thus requires
– Mechanisms to limit memory access to protected memory
– At least two modes of execution for instructions
• Privileged mode: an instruction is allowed to do what it whatever it wants -> kernel mode for OS
• Non-privileged mode: user-level processes
• Intel x86 Architecture: processor supports four levels
– Level 0 used by OS
– Level 3 used by regular applications
•Provide mechanisms to go from non-privileged mode to privileged mode -> system call
•Provide a portion of processor state that a user process can read but not modify
• E.g. memory protection information
• Each guest OS maintains its page tables to do the mapping from virtual address to physical address
• Most simple solution: VMM holds an additional table which maps the physical address of a guest OS onto the 'machine address'
– Introduces a third level of redirection for every memory access
• Alternative solution: VMM maintains a shadow page table of each guest OS
– Copy of the page table of the OS
– Page tables still works with regular physical addresses
– Only modifications to the page table are intercepted by the VMM

3. Explain the following advanced optimization of cache:

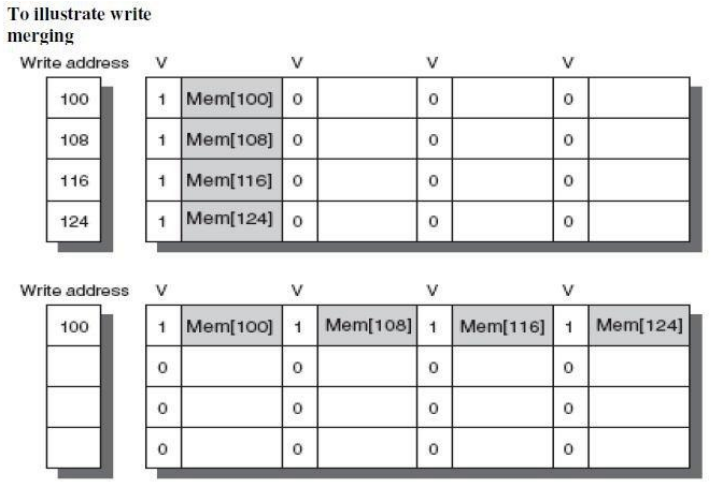
1.) Compiler optimizations to reduce miss rate.
2.) Merging write buffer to reduce miss penalty.
3.) Non blocking cache to increase cache band-width.

(June2013) (June 2015)(Jan 2016)

. Merging Write Buffer to Reduce Miss Penalty

•Write buffer to allow processor to continue while waiting to write to memory
•If buffer contains modified blocks, the addresses can be checked to see if address

of new data matches the address of a valid write buffer entry -If so, new data are
combined with that entry
•Increases block size of write for write-through cache of writes to sequential
words, bytes since multiword writes more efficient to memory
•The Sun T1 (Niagara) processor, among many others, uses write merging



Reducing Misses by Compiler Optimizations

•McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte
blocks in software

• **Instructions**

– Reorder procedures in memory so as to reduce conflict misses
– Profiling to look at conflicts (using tools they developed)

• **Data**

– Merging Arrays: improve spatial locality by single array of compound elements vs. 2
arrays
– Loop Interchange: change nesting of loops to access data in order
stored in memory

– Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap
– Blocking: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

Compiler Optimizations- Reduction comes from software (no Hw ch.)

Loop Interchange

•Motivation: some programs have nested loops that access data in nonsequential order
•Solution: Simply exchanging the nesting of the loops can make the code access the data in the order it is stored =>
reduce misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded

Loop Interchange Example

```
/* Before */
for (j = 0; j < 100; j = j+1)
for (i = 0; i < 5000; i = i+1)
x[i][j] = 2 * x[i][j];
/* After */
for (i = 0; i < 5000; i = i+1)
for (j = 0; j < 100; j = j+1)
x[i][j] = 2 * x[i][j];
```

Blocking

•Motivation: multiple arrays, some accessed by rows and some by columns

•Storing the arrays row by row (row major order) or column by column (column major order) does not help: both rows and columns are used in every iteration of the loop (Loop Interchange cannot help)

•Solution: instead of operating on entire rows and columns of an array, blocked algorithms operate on submatrices or blocks => maximize accesses to the data loaded into the cache before the data is replaced

4. Explain internal organization of 64 Mb DRAM. (June/July 2014) (June 2013) (Dec 2013) (Jan 2015)(June 2016)

DRAM Technology

• Semiconductor Dynamic Random Access Memory
• Emphasize on cost per bit and capacity
• Multiplex address lines ı cutting # of address pins in half
– Row access strobe (RAS) first, then column access strobe (CAS)
– Memory as a 2D matrix – rows go to a buffer
– Subsequent CAS selects subrow
• Use only a single transistor to store a bit
– Reading that bit can destroy the information
– Refresh each bit periodically (ex. 8 milliseconds) by writing back
• Keep refreshing time less than 5% of the total time
• DRAM capacity is 4 to 8 times that of SRAM

• DIMM: Dual inline memory module
– DRAM chips are commonly sold on small boards called DIMMs
– DIMMs typically contain 4 to 16 DRAMs
• Slowing down in DRAM capacity growth
– Four times the capacity every three years, for more than 20 years
– New chips only double capacity every two year, since 1998
• DRAM performance is growing at a slower rate
– RAS (related to latency): 5% per year
– CAS (related to bandwidth): 10%+ per year

RAS improvement

SRAM Technology

• Cache uses SRAM: Static Random Access Memory
• SRAM uses six transistors per bit to prevent the information from being disturbed when read
_no need to refresh
– SRAM needs only minimal power to retain the charge in the standby mode _ good for embedded applications
– No difference between access time and cycle time for SRAM
• Emphasize on speed and capacity
– SRAM address lines are not multiplexed
• SRAM speed is 8 to 16x that of DRAM

Improving Memory Performance
in a Standard DRAM Chip
• Fast page mode: time signals that allow repeated accesses to buffer without another row access time
• Synchronous RAM (SDRAM): add a clock signal to DRAM interface, so that the repeated transfer would not bear overhead to synchronize with the controller
– Asynchronous DRAM involves overhead to sync with controller
– Peak speed per memory module 800—1200MB/sec in 2001
• Double data rate (DDR): transfer data on both the rising edge and falling edge of DRAM clock signal
– Peak speed per memory module 1600—2400MB/sec in 2001

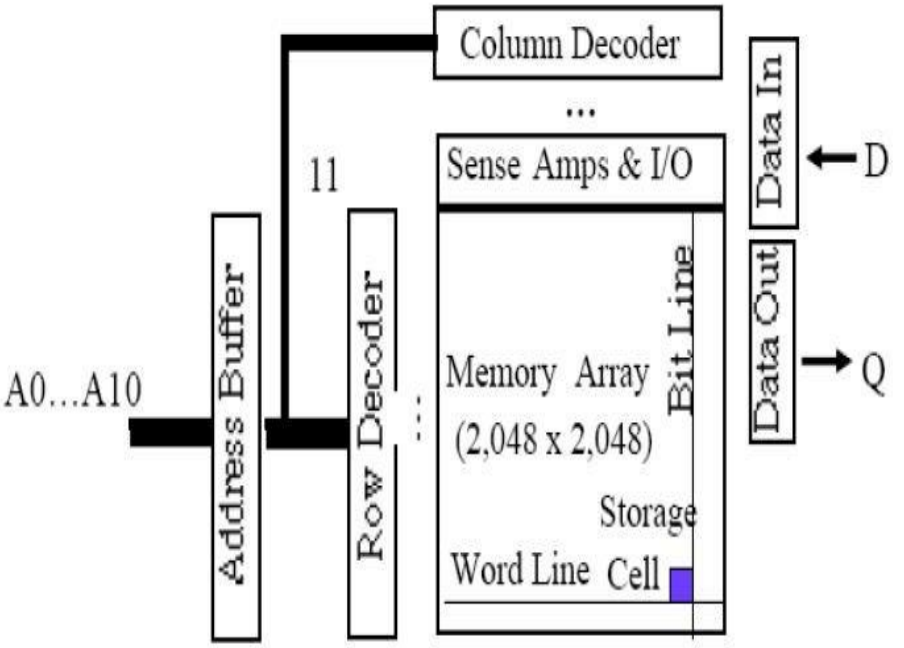


Fig: Internal organization of 64 MB DRAM

UNIT 8

1. Explain in detail the hardware support for preserving exception behavior during Speculation. July 14) (June 2013) (June 2015)(Jan 2016)(June 2016)

H/W Support : Conditional Execution

• Also known as Predicated Execution
  – Enhancement to instruction set
  – Can be used to eliminate branches
  – All control dependences are converted to data dependences
• Instruction refers to a condition
  – Evaluated as part of the execution
• True?
  – Executed normally
• False?
  – Execution continues as if the instruction were a no-op
• Example :
  – Conditional move between registers
  –

Example

```
if (A==0)
S = T;
Straightforward Code
BNEZ   R1,      L;
ADDU R2, R3, R0
L:
Conditional Code
CMOVZ R2, R3, R1
```

Annulled if R1 is not 0

Conditional Instruction …

• Can convert control to data dependence
• In vector computing, it's called *if conversion*.
• Traditionally, in a pipelined system
  – Dependence has to be resolved closer to front of pipeline
• For conditional execution
  – Dependence is resolved at end of pipeline, closer to the register write

Another example

• A = abs(B)

```
if (B < 0)
A = -B;
else
A = B;
```

• Two conditional moves

One unconditional and one conditional move
• The branch condition has moved into the instruction
– Control dependence becomes data dependence
–

Limitations of Conditional Moves
• Conditional moves are the simplest form of predicated instructions
• Useful for short sequences
• For large code, this can be inefficient
– Introduces many conditional moves
• Some architectures support full predication
– All instructions, not just moves
• Very useful in global scheduling
– Can handle nonloop branches nicely
– Eg : The whole if portion can be predicated if the frequent path is not taken

Example

LW	R1, 40(R2)	ADD	R3,R4,R5
		ADD	R6, R3, R7
BEQZ	R10, L		
LW	R8, 0(R10)		
LW	R9, 0(R8)		

• Assume : Two issues, one to ALU and one to memory; or branch by itself
• Wastes a memory operation slot in second cycle
• Can incur a data dependence stall if branch is not taken
– R9 depends on R8

Predicated Execution
Assume : LWC is predicated load and loads if third operand is not 0

LW	R1, 40(R2)	ADD	R3,R4,R5
LWC	R8, 0(R10), R10	ADD	R6, R3, R7
BEQZ	R10, L		
LW	R9, 0(R8)		

• One instruction issue slot is eliminated
• On mispredicted branch, predicated instruction will not have any effect
• If sequence following the branch is short, the entire block of the code can be predicated

Some Complications

- Exception Behavior
  - – Must not generate exception if the predicate is false
- If R10 is zero in the previous example
  - – LW R8, 0(R10) can cause a protection fault
- If condition is satisfied
  - – A page fault can still occur
- Biggest Issue – Decide when to annul an instruction
  - – Can be done during issue
- Early in pipeline
- Value of condition must be known early, can induce stalls
  - – Can be done before commit
- Modern processors do this
- Annulled instructions will use functional resources
- Register forwarding and such can complicate implementation

2. Explain the prediction and Speculation support provided in IA64? (Dec 2013) (June 2014)

IA-64 Pipeline Features

•Branch Prediction
  –Predicate Registers allow branches to be turned on or off
  –Compiler can provide branch prediction hints
•Register Rotation
  –Allows faster loop execution in parallel
•Predication Controls Pipeline Stages

Cache Features

•L1 Cache
  –4 way associative
  –*16Kb* Instruction
  –16Kb Data
•L2 Cache
  –*Itanium*
•6 way associative
•96 Kb
  –*Itanium2*
•8 way associative
•256 Kb Initially
  –256Kb Data and 1Mb Instruction on Montvale!

Cache Features

•L3 Cache
  –*Itanium*
•4 way associative
•Accessible through FSB
•2-4Mb

–*Itanium2*

•2 – 4 way associative

•On Die

•3Mb

–Up to 24Mb on Montvale chips(12Mb/core)!

Register Specification

_128, 65-bit General Purpose Registers

_128, 82-bit Floating Point Registers

_128, 64-bit Application Registers

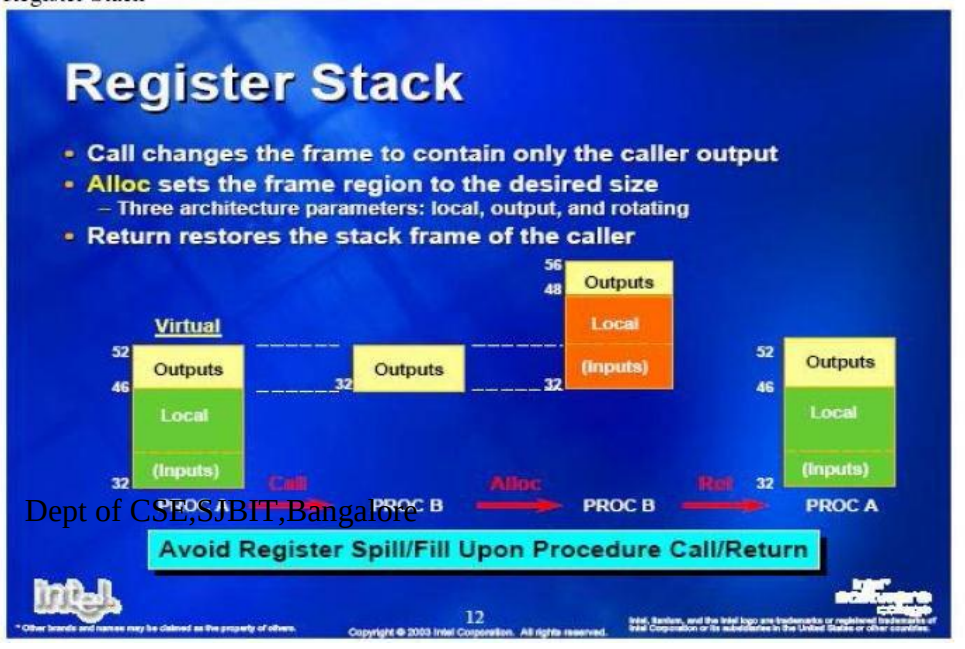
_8, 64-bit Branch Registers

_64, 1-bit Predicate Registers

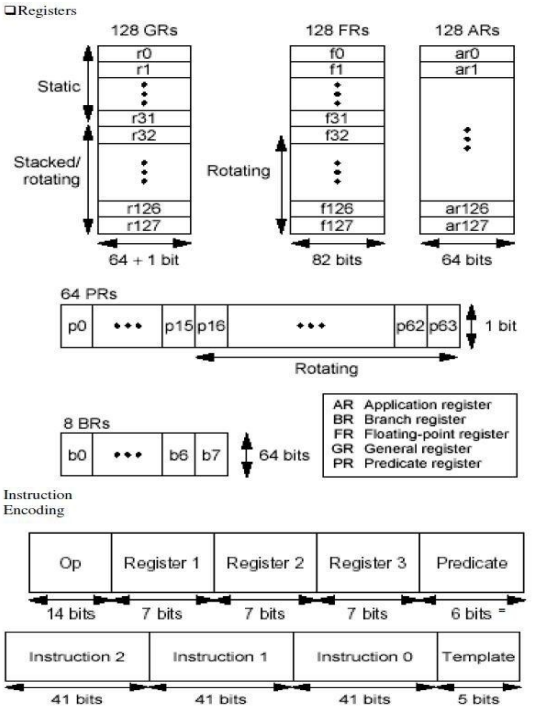
Register Model

- _128 General and Floating Point Registers
- _32 always available, 96 on stack
- _As functions are called, compiler allocates a specific number of local and output
- registers to use in the function by using register allocation instruction "Alloc".
- _Programs renames registers to start from 32 to 127.
- _Register Stack Engine (RSE) automatically saves/restores stack to memory when needed
- _RSE may be designed to utilize unused memory bandwidth to perform register spill and fill operations in the background

Register Stack



On function call, machine shifts register window such that previous output registers become new locals starting at r32



Five execution unit slots

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

Possible Template Values

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F

Figure G.7 The 24 possible template values (8 possible values are reserved) and the instruction slots and stops for each format. Stops are

Straightforward MIPS code

```
Loop:   L.D     F0,0(R1)    ;F0=array element
        ADD.D   F4,F0,F2    ;add scalar in F2
        S.D     F4,0(R1)    ;store result

        DADDUI  R1,R1,#-8   ;decrement pointer
                            ;8 bytes (per DW)
        BNE     R1,R2,Loop  ;branch R1!=R2
```

The code scheduled to minimize the number of bundles

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/cycle)
9: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		1
14: M M F	L.D F10,-16(R1)	L.D F14,-24(R1)	ADD.D F4,F0,F2	3
15: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F8,F6,F2	4
15: M M F	L.D F26,-48(R1)	S.D F4,0(R1)	ADD.D F12,F10,F2	6
15: M M F	S.D F8,-8(R1)	S.D F12,-16(R1)	ADD.D F16,F14,F2	9
15: M M F	S.D F16,-24(R1)		ADD.D F20,F18,F2	12
15: M M F	S.D F20,-32(R1)		ADD.D F24,F22,F2	15
15: M M F	S.D F24,-40(R1)		ADD.D F28,F26,F2	18
16: M I B	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop	21

(a) The code scheduled to minimize the number of bundles

The code scheduled to minimize the number of cycles assuming one bundle executed per cycle

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/cycle)
8: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		1
9: M M I	L.D F10,-16(R1)	L.D F14,-24(R1)		2
14: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	3
14: M M F	L.D F26,-48(R1)		ADD.D F8,F6,F2	4
15: M M F			ADD.D F12,F10,F2	5
14: M M F		S.D F4,0(R1)	ADD.D F16,F14,F2	6
14: M M F		S.D F8,-8(R1)	ADD.D F20,F18,F2	7
15: M M F		S.D F12,-16(R1)	ADD.D F24,F22,F2	8
14: M M F		S.D F16,-24(R1)	ADD.D F28,F26,F2	9
9: M M I	S.D F20,-32(R1)	S.D F24,-40(R1)		11
16: M I B	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop	12

(b) The code scheduled to minimize the number of cycles assuming one bundle executed per cycle

Unrolled loop after it has been scheduled for the pipeline

```
Loop:   L.D     F0,0(R1)
        L.D     F6,-8(R1)
        L.D     F10,-16(R1)
        L.D     F14,-24(R1)
ADD.D   F4,F0,F2
ADD.D   F8,F6,F2
ADD.D   F12,F10,F2
ADD.D   F16,F14,F2
S.D     F4,0(R1)
S.D     F8,-8(R1)
DADDUI  R1,R1,#-32
S.D     F12,16(R1)
S.D     F16,8(R1)
BNE     R1,R2,Loop
```

Instruction Encoding

- **Each instruction includes the opcode and three operands**
- **Each instructions holds the identifier for a corresponding Predicate Register**
- **Each bundle contains 3 independent instructions**
- **Each instruction is 41 bits wide**
- **Each bundle also holds a 5 bit template field**

Distributing Responsibility

_ILP Instruction Groups

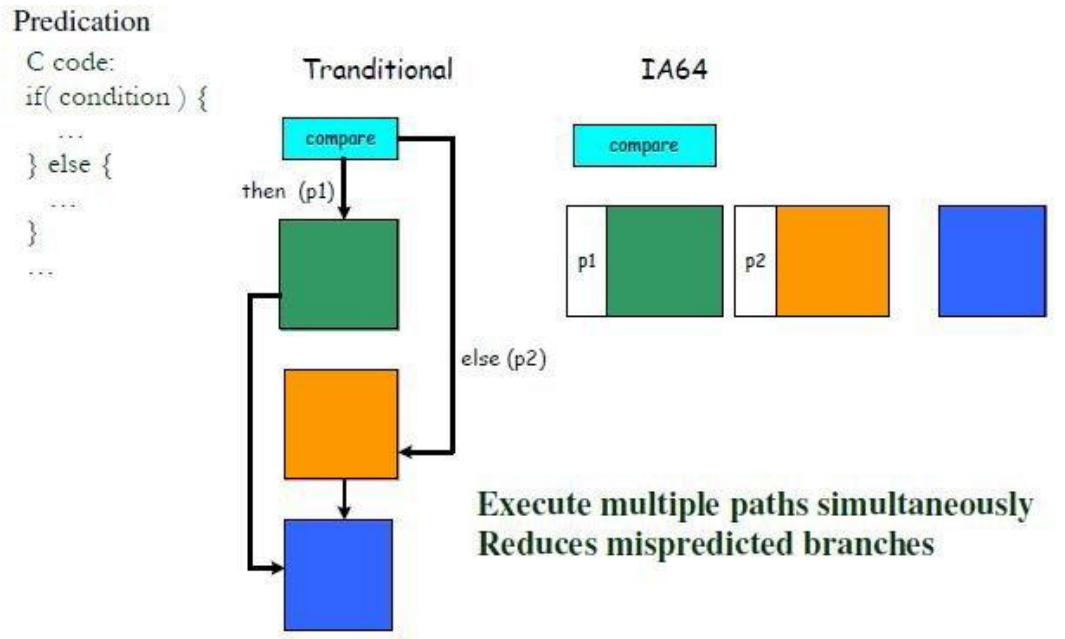
_Control flow parallelism

Parallel comparison

Multiway branches

_Influencing dynamic events

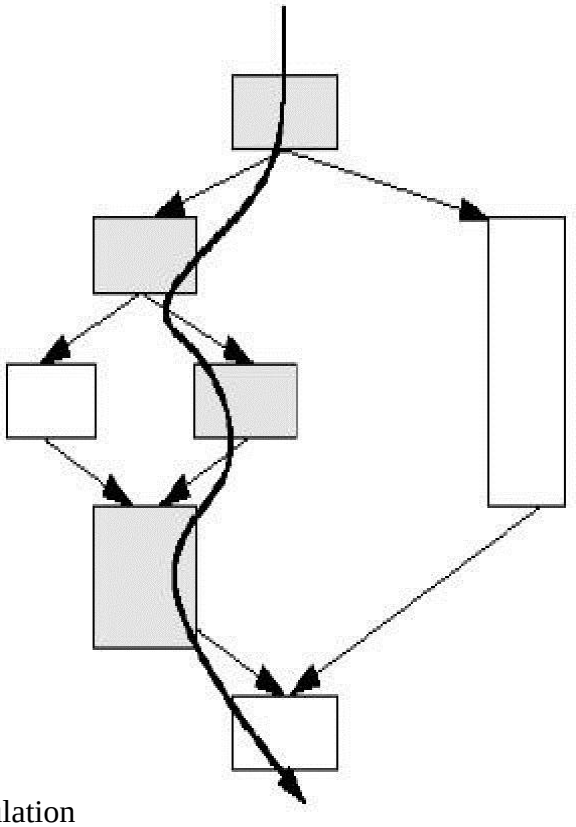
Provides an extensive set of hints that the compiler uses to tell the hardware about likely branch behavior (taken or not taken, amount to fetch at branch target) and memory operations (in what level of the memory hierarchy to cache data).



- _Use predicates to eliminate branches, move instructions across branches
- _Conditional execution of an instruction based on predicate register (64 1-bit predicate registers)
- _Predicates are set by compare instructions
- _Most instructions can be predicated – each instruction code contains predicate field
- _If predicate is true, the instruction updates the computation state; otherwise, it behaves like a nop
-

Scheduling and Speculation

• Basic block: code with single entry and exit, exit point can be multiway branch
• Control Improve ILP by statically move ahead long latency code blocks.
• path is a frequent execution path
• Schedule for control paths
• Because of branches and loops, only small percentage of code is executed regularly
• Analyze dependences in blocks and paths
• Compiler can analyze more efficiently - more time, memory, larger view of the program
• Compiler can locate and optimize the commonly executed blocks



Control speculation

_ Not all the branches can be removed using predication.

_ Loads have longer latency than most instructions and tend to start timecritical chains of instructions

_ Constraints on code motion on loads limit parallelism

_ Non-EPIC architectures constrain motion of load instruction

_ IA-64: Speculative loads, can safely schedule load instruction before one or more prior branches

Control Speculation

_Exceptions are handled by setting NaT (Not a Thing) in target register

_Check instruction-branch to fix-up code if NaT flag set

_Fix-up code: generated by compiler, handles exceptions

_NaT bit propagates in execution (almost all IA-64 instructions)

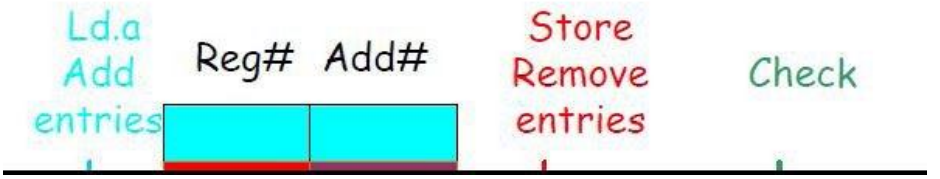
_NaT propagation reduces required check points

Speculative Load

_ Load instruction (ld.s) can be moved outside of a basic block even if branch target is not known

_ Speculative loads does not produce exception - it sets the NaT
_ Check instruction (chk.s) will jump to fix-up code if NaT is set
Data Speculation
_ The compiler may not be able to determine the location in memory being referenced (pointers)
_ Want to move calculations ahead of a possible memory dependency
_ Traditionally, given a store followed by a load, if the compiler cannot determine if the addresses will be equal, the load cannot be moved ahead of the store.
_ IA-64: allows compiler to schedule a load before one or more stores
_ Use advance load (ld.a) and check (chk.a) to implement
_ ALAT (Advanced Load Address Table) records target register, memory address accessed, and access size

Data Speculation
1. Allows for loads to be moved ahead of stores even if the compiler is unsure if addresses are the same
2. A speculative load generates an entry in the ALAT
3. A store removes every entry in the ALAT that have the same address
4. Check instruction will branch to fix-up if the given address is not in the ALAT



• **Use address field as the key for comparison**
• If an address cannot be found, run recovery code
• ALAT are smaller and simpler implementation than equivalent structures for superscalars

3.Explain in detail the hardware support for preserving exception behavior during speculation. (July 2014) (June2013) (June 2016)

H/W Support : Conditional Execution
• Also known as Predicated Execution
– Enhancement to instruction set
– Can be used to eliminate branches
– All control dependences are converted to data dependences
• Instruction refers to a condition
– Evaluated as part of the execution

• True?
– Executed normally
• False?
– Execution continues as if the instruction were a no-op
• Example :
– Conditional move between registers
–

Example

if (A==0)
S = T;
Straightforward Code
BNEZ R1, L;
ADDU R2, R3, R0
L:
Conditional Code
CMOVZ R2, R3, R1
Annulled if R1 is not 0
Conditional Instruction …
• Can convert control to data dependence
• In vector computing, it's called *if conversion*.
• Traditionally, in a pipelined system
– Dependence has to be resolved closer to front of pipeline
• For conditional execution
– Dependence is resolved at end of pipeline, closer to the register write

Another example
• A = abs(B)
if (B < 0)
A = -B;
else
A = B;
• Two conditional moves
• One unconditional and one conditional move
• The branch condition has moved into the instruction
– Control dependence becomes data dependence

Limitations of Conditional Moves

• Conditional moves are the simplest form of predicated instructions
• Useful for short sequences
• For large code, this can be inefficient
– Introduces many conditional moves

• Some architectures support full predication
– All instructions, not just moves
• Very useful in global scheduling
– Can handle nonloop branches nicely
– Eg : The whole if portion can be predicated if the frequent path is not taken

Example

LW	R1, 40(R2)	ADD	R3,R4,R5
		ADD	R6, R3, R7
BEQZ	R10, L		
LW	R8, 0(R10)		
LW	R9, 0(R8)		

• Assume : Two issues, one to ALU and one to memory; or branch by itself
• Wastes a memory operation slot in second cycle
• Can incur a data dependence stall if branch is not taken
– R9 depends on R8

Predicated Execution

Assume : LWC is predicated load and loads if third operand is not 0

LW	R1, 40(R2)	ADD	R3,R4,R5
LWC	R8, 0(R10), R10	ADD	R6, R3, R7
BEQZ	R10, L		
LW	R9, 0(R8)		

• One instruction issue slot is eliminated
• On mispredicted branch, predicated instruction will not have any effect
• If sequence following the branch is short, the entire block of the code can be predicated

Some Complications

• Exception Behavior
– Must not generate exception if the predicate is false
• If R10 is zero in the previous example

– LW R8, 0(R10) can cause a protection fault
• If condition is satisfied
– A page fault can still occur
• Biggest Issue – Decide when to annul an instruction
– Can be done during issue
• Early in pipeline
• Value of condition must be known early, can induce stalls
– Can be done before commit
• Modern processors do this
• Annulled instructions will use functional resources
• Register forwarding and such can complicate implementation

4. Explain the architecture of IA64 Intel processor and also the prediction and speculation support provided. (June 2013) (Jan 2014) (Jan 2015) (June 2015)(Jan 2016)

IA-64 and Itanium Processor

Introducing The IA-64 Architecture

Itanium and Itanium2 Processor

Slide Sources: Based on "Computer Architecture" by Hennessy/Patterson.
Supplemented from various freely downloadable sources
IA-64 is an EPIC

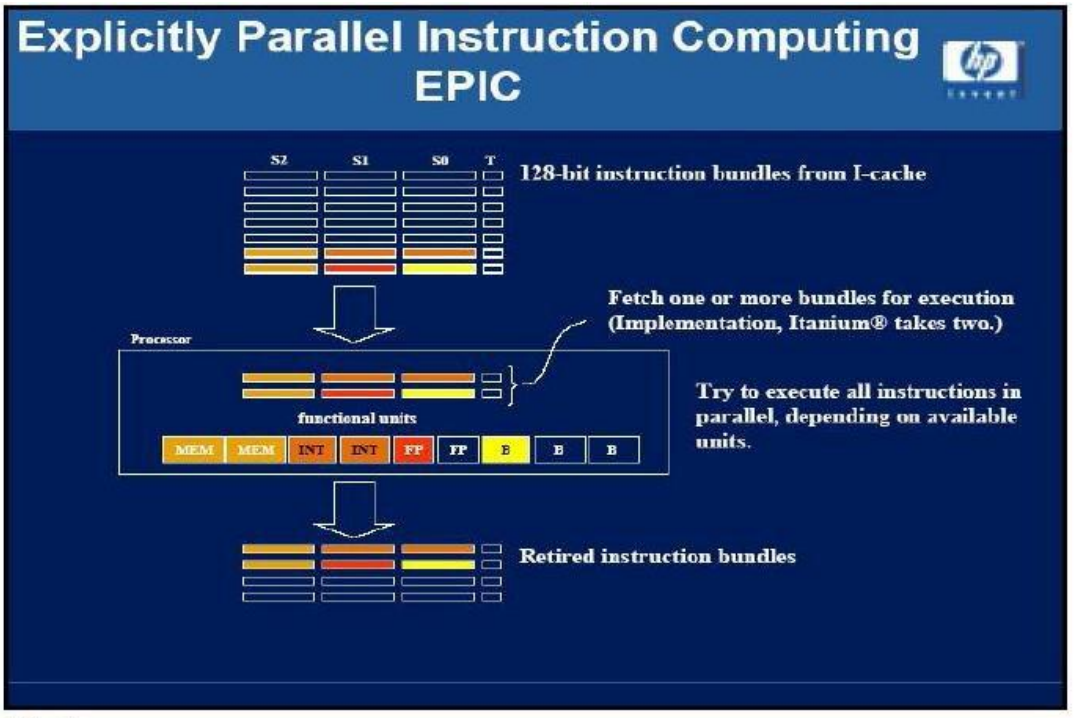
•IA-64 largely depends on software for parallelism
• VLIW – Very Long Instruction Word
• EPIC – Explicitly Parallel Instruction Computer
VLIW points
•VLIW – Overview
– RISC technique
– Bundles of instructions to be run in parallel
– Similar to superscaling
– Uses compiler instead of branch prediction hardware

EPIC
•EPIC – Overview
– Builds on VLIW
– Redefines instruction format
– Instruction coding tells CPU how to process data
– Very compiler dependent
– Predicated execution

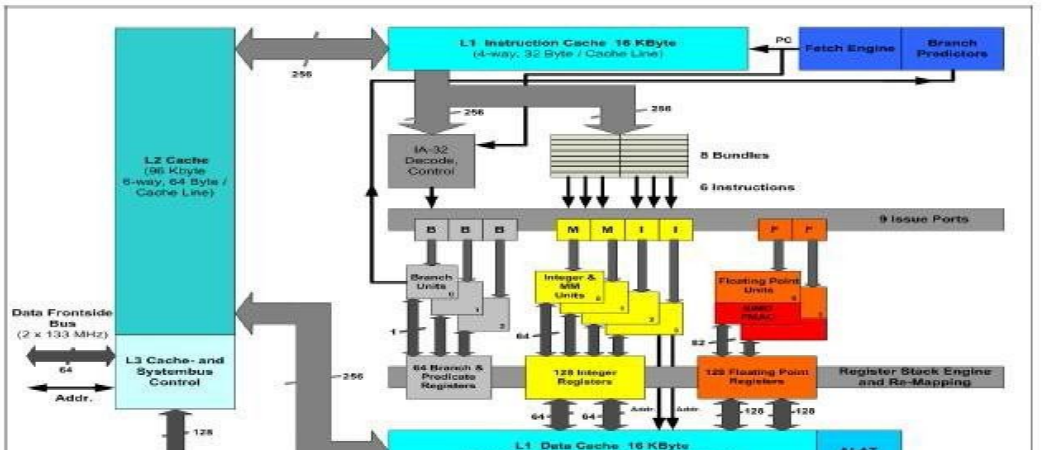
EPIC pros and cons

•EPIC – Pros:
– Compiler has more time to spend with code

– Time spent by compiler is a one-time cost
- **Reduces circuit complexity**



Chip Layout
•*Itanium* Architecture Diagram

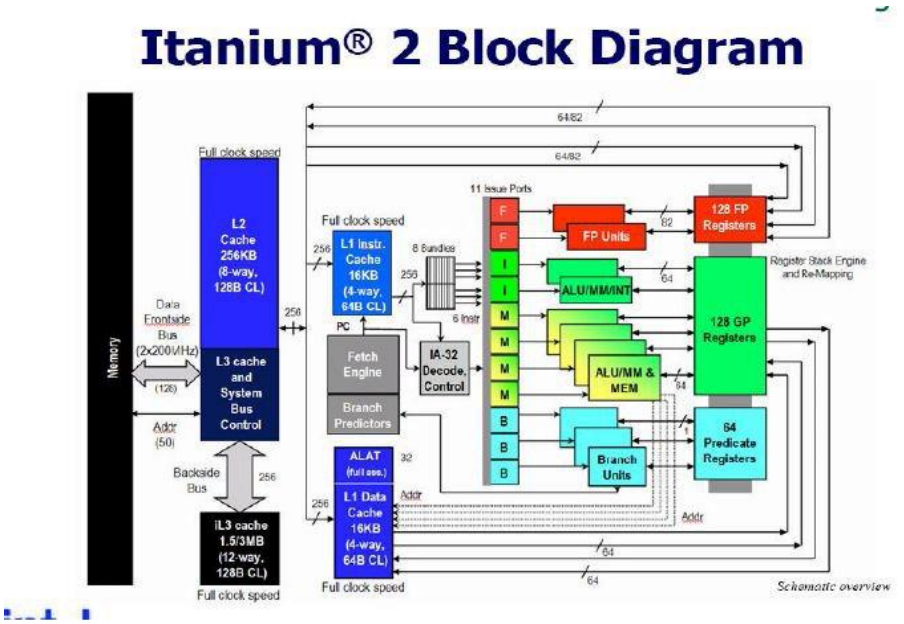


Itanium Specs

- •4 Integer ALU's
- •4 multimedia ALU's
- •2 Extended Precision FP Units
- •2 Single Precision FP units
- •2 Load or Store Units
- •3 Branch Units
- •10 Stage 6 Wide Pipeline
- •32k L1 Cache
- •96K L2 Cache
- •4MB L3 Cache(extern)þ
- •800Mhz Clock

Intel Itanium

•800 MHz
•10 stage pipeline
•Can issue 6 instructions (2 bundles) per cycle
•4 Integer, 4 Floating Point, 4 Multimedia, 2 Memory, 3 Branch Units
•32 KB L1, 96 KB L2, 4 MB L3 caches
•2.1 GB/s memory bandwidth



Itanium2 Specs

•6 Integer ALU's

•6 multimedia ALU's

•2 Extended Precision FP Units
•2 Single Precision FP units
•2 Load and Store Units
•3 Branch Units
•8 Stage 6 Wide Pipeline
•32k L1 Cache
•256K L2 Cache
•3MB L3 Cache(on die)þ
•1Ghz Clock initially
–Up to 1.66Ghz on Montvale

Itanium2 Improvements
•Initially a 180nm process
–Increased to 130nm in 2003
–Further increased to 90nm in 2007
•Improved Thermal Management
•Clock Speed increased to 1.0Ghz
•Bus Speed Increase from 266Mhz to 400Mhz

•L3 cache moved on die
–Faster access rate

IA-64 Pipeline Features
•Branch Prediction
–Predicate Registers allow branches to be turned on or off
–Compiler can provide branch prediction hints
•Register Rotation
–Allows faster loop execution in parallel
•Predication Controls Pipeline Stages
Cache Features
•L1 Cache
–4 way associative
–*16Kb* Instruction
–16Kb Data
•L2 Cache
–*Itanium*
•6 way associative
•96 Kb
–*Itanium2*
•8 way associative
•256 Kb Initially
–256Kb Data and 1Mb Instruction on Montvale!
Cache Features

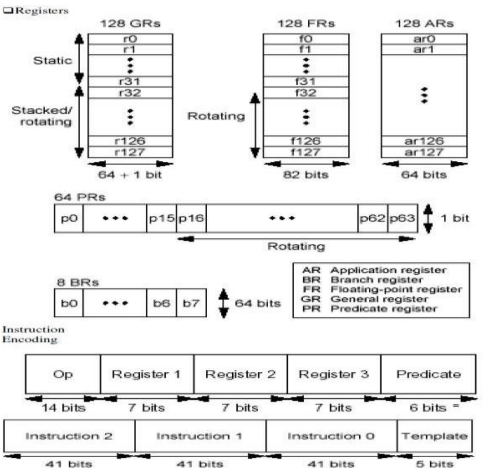
•L3 Cache
–*Itanium*

•4 way associative
•Accessible through FSB
•2-4Mb
–*Itanium2*
•2 – 4 way associative
•On Die
•3Mb
–Up to 24Mb on Montvale chips(12Mb/core)!

Register
Specification
_128, 65-bit General Purpose Registers
_128, 82-bit Floating Point Registers
_128, 64-bit Application Registers
_8, 64-bit Branch Registers
_64, 1-bit Predicate Registers

Register Model
- _128 General and Floating Point Registers
- _32 always available, 96 on stack
- _As functions are called, compiler allocates a specific number of local and output
- registers to use in the function by using register allocation instruction "Alloc".
- _Programs renames registers to start from 32 to 127.
- _Register Stack Engine (RSE) automatically saves/restores stack to memory when needed
- _RSE may be designed to utilize unused memory bandwidth to perform register spill and fill operations in the background

On function call, machine shifts register window such that previous output registers become new locals starting at



Five execution unit slots

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

Possible Template Values

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F

Figure G.7 The 24 possible template values (8 possible values are reserved) and the instruction slots and stops for each format. Stops are indicated by heavy lines

5. Consider the loop below:
For (i = 1;i ≤ 100 ; i = i+1 {
A[i]= A[i] + B[i] ; 1 * S1 *1
B[i+1]= C[i] + D[i] ; 1 * S2 *1
}
What are the dependencies between S1 and S2 ? Is the loop parallel ? If not show how to make it parallel. (Dec 2013) (Jan 2015)(June 2016)

•For the loop:
–for (i=1; i<=100; i=i+1) { A[i+1] = A[i] + C[i]; /* S1 */
B[i+1] = B[i] + A[i+1]; /* S2 */ }

–what are the dependences?
•There are two different dependences:

–**loop-carried: (prevents parallel operation of iterations)**
•S1 computes A[i+1] using value of A[i] computed in previous iteration
•S2 computes B[i+1] using value of B[i] computed in previous iteration
–**not loop-carried: (parallel operation of iterations is ok)**
•S2 uses the value A[i+1] computed by S1 in the *same* iteration
•The loop-carried dependences in this case *force* successive iterations of the loop to execute in series. *Why?*
–S1 of iteration *i* depends on S1 of iteration *i-1* which in turn depends on …, etc.
Another Loop with Dependences
•Generally, loop-carried dependences *hinder* ILP
–if there are no loop-carried dependences all iterations could be executed in parallel
–even if there are loop-carried dependences it may be possible to parallelize the loop – an analysis of the dependences is required…

•**For the loop:**
–**for (i=1; i<=100; i=i+1) { A[i] = A[i] + B[i]; /* S1 */**
B[i+1] = C[i] + D[i]; /* S2 */ }

–***what are the dependences?***
•There is one loop-carried dependence:
–S1 uses the value of B[i] computed in a previous iteration by S2
–but this *does not force* iterations to execute in series. *Why…?*
–…because S1 of iteration *i* depends on S2 of iteration *i-1*…, and the *chain of dependences stops here*!

Parallelizing Loops with Short Chains of Dependences

•Parallelize the loop:
–for (i=1; i<=100; i=i+1) { A[i] = A[i] + B[i]; /* S1 */
B[i+1] = C[i] + D[i]; /* S2 */ }
•Parallelized code:
–A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1)
{ B[i+1] = C[i] + D[i];
A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
–the dependence between the two statements in the loop is no longer loop-carried and iterations of the loop may be executed in parallel
–