P.R ENGINEERING COLLEGE

VALLAM-THANJAVUR

CS-2354 ADVANCED COMPUTER ARCHITECTURE

Prepared by

Ms.R..Arivumalar
Asst.prof
Dept of CSE
P.R Engineering College
Vallam-Thanjavur

CS2354 ADVANCED COMPUTER ARCHITECTURE

UNIT I INSTRUCTION LEVEL PARALLELISM

ILP – Concepts and challenges – Hardware and software approaches – Dynamic scheduling – Speculation - Compiler techniques for exposing ILP – Branch prediction.

UNIT II MULTIPLE ISSUE PROCESSORS

VLIW & EPIC – Advanced compiler support – Hardware support for exposing parallelism – Hardware versus software speculation mechanisms – IA 64 and Itanium processors – Limits on ILP.

UNIT III MULTIPROCESSORS AND THREAD LEVEL PARALLELISM

Symmetric and distributed shared memory architectures – Performance issues – Synchronization – Models of memory consistency – Introduction to Multithreading.

UNIT IV MEMORY AND I/O

Cache performance – Reducing cache miss penalty and miss rate – Reducing hit time – Main memory and performance – Memory technology. Types of storage devices – Buses – RAID – Reliability, availability and dependability – I/O performance measures – Designing an I/O system.

UNIT V MULTI-CORE ARCHITECTURES

Software and hardware multithreading – SMT and CMP architectures – Design issues – Case studies – Intel Multi-core architecture – SUN CMP architecture - heterogeneous multi-core processors – case study: IBM Cell Processor.

TEXT BOOKS:

1. John L. Hennessey and David A. Patterson, " Computer architecture – A quantitative approach", Morgan Kaufmann / Elsevier Publishers, 4th. edition, 2007.

REFERENCES:

1. David E. Culler, Jaswinder Pal Singh, "Parallel computing architecture: A hardware/software approach" , Morgan Kaufmann /Elsevier Publishers, 1999.
2. Kai Hwang and Zhi.Wei Xu, "Scalable Parallel Computing", Tata McGraw Hill, New Delhi, 2003.

Unit No: 1

UNIT I INSTRUCTION LEVEL PARALLELISM

ILP – Concepts and challenges – Hardware and software approaches – Dynamic scheduling – Speculation - Compiler techniques for exposing ILP – Branch prediction.

Instruction Level Parallelism

Instruction-Level Parallelism: Concepts and Challenges:

- Instruction-level parallelism (ILP) is the potential overlap the execution of instructions using Pipeline concept to improve performance of the system.
- The various techniques that are used to Increase amount of parallelism are reduces the impact of data and control hazards and increases Processor ability to exploit parallelism

There are two approaches to exploiting ILP.

1. Static Technique – Software Dependent

2. Dynamic Technique – Hardware Dependent

Technique	Reduces
Forwarding and bypassing	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Basic dynamic scheduling (score boarding)	Data hazard stalls from true dependences
Dynamic scheduling with renaming	Data hazard stalls and stalls from anti dependences and output dependences
Dynamic branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Speculation	Data hazard and control hazard stalls
Dynamic memory disambiguation	Data hazard stalls with memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis	Ideal CPI, data hazard stalls
Software pipelining, trace scheduling	Ideal CPI, data hazard stalls
Compiler speculation	Ideal CPI, data, control stalls

- The simplest and most common way to increase the amount of parallelism is loop-level parallelism.

- Here is a simple example of a loop, which adds two 1000-element arrays, that is completely parallel:
- for (i=1;i<=1000; i=i+1) x[i] = x[i] + y[i];
- CPI (Cycles per Instruction) for a pipelined processor is the sum of the base CPI and all Contributions from stalls:
- Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls
- The ideal pipeline CPI is a measure of the maximum performance attainable by the implementation.
- By reducing each of the terms of the right-hand side, we minimize the overall pipeline CPI and thus increase the IPC (Instructions per Clock).

Various types of Dependences in ILP.

Data Dependence and Hazards

- To exploit instruction-level parallelism, determine which instructions can be executed in parallel. If two instructions are parallel, they can execute simultaneously in a pipeline without causing any stalls.
- If two instructions are dependent they are not parallel and must be executed in order.

There are three different types of dependences:

- Data dependences (also called true data dependences), name dependences, and control dependences.

Data Dependences

- An instruction j is data dependent on instruction i if either of the following holds:
  - Instruction i produces a result that may be used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions.
- This dependence chain can be as long as the entire program.

- For example, consider the following code sequence that increments a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2:
- 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 (/e BNE R1,R2,LOOP ; branch R1!=zero

- The dependence implies that there would be a chain of one or more data hazards between the two instructions.

- Executing the instructions simultaneously will cause a processor with pipeline interlocks to detect a hazard and stall, thereby reducing or eliminating the overlap. Dependences are a property of programs.

- The presence of the dependence indicates the potential for a hazard, but the actual hazard and the length of any stall is a property of the pipeline. The importance of the data dependences is that a dependence

- indicates the possibility of a hazard,
- Determines the order in which results must be calculated, and
- Sets an upper bound on how much parallelism can possibly be exploited.

Name Dependences

- The name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name.

- There are two types of name dependences between an instruction i that precedes instruction j in program order:

    - An antidependence between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i read the correct value.

    - An output dependence occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

- Both anti-dependences and output dependences are name dependences, as opposed to true data dependences, since there is no value being transmitted between the instructions.
- Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

- This renaming can be more easily done for register operands, where it is called register renaming. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

Control Dependences

- A control dependence determines the ordering of an instruction, i, with respect to a branch instruction so that the instruction i is executed in correct program order.
- Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order.
- One of the simplest examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch. For example, in the code segment:

```
if p1 { S1;
};
if p2 { S2;
}
```

- S1 is control dependent on p1, and S2is control dependent on p2 but not on p1. In general, there are two constraints imposed by control dependences:

- An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch. For example, we cannot take an instruction from the then-portion of an if-statement and move it before the if- statement.

- An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch. For example, we cannot take a statement before the if-statement and move it into the then-portion.

- Control dependence is preserved by two properties in a simple pipeline, First, instructions execute in program order.
- This ordering ensures that an instruction that occurs before a branch is executed before the branch. Second, the detection of control or branch hazards ensures that an instruction that is control dependent on a branch is not executed until the branch direction is known.

Data Hazard and various hazards in ILP.

Data Hazards

- A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap caused by pipelining, or other reordering of instructions, would change the order of access to the operand involved in the dependence.

- Because of the dependence, preserve order that the instructions would execute in, if executed sequentially one at a time as determined by the original source program.
- The goal of both our software and hardware techniques is to exploit parallelism by preserving program order only where it affects the outcome of the program.
- Detecting and avoiding hazards ensures that necessary program order is preserved.

- Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions.
- Consider two instructions i and j, with i occurring before j in program order. The possible data hazards are RAW (read after write) — j tries to read a source before i writes it, so j incorrectly gets the old value. This hazard is the most common type and corresponds to a true data dependence.
- Program order must be preserved to ensure that j receives the value from i. In the simple common five-stage static pipeline a load instruction followed by an integer ALU instruction that directly uses the load result will lead to a RAW hazard.

- WAW (write after write) — j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard corresponds to output dependence.
- WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

- The classic five-stage integer pipeline writes a register only in the WB stage and avoids this class of hazards.
- WAR (write after read) — j tries to write a destination before it is read by i, so i incorrectly gets the new value. This hazard arises from an antidependence.
- WAR hazards cannot occur in most static issue pipelines even deeper pipelines or floating point pipelines because all reads are early (in ID) and all writes are late (in WB).
- A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline or when instructions are reordered.

Dynamic Scheduling

Overcoming Data Hazards with Dynamic Scheduling

- The Dynamic Scheduling is used handle some cases when dependences are unknown at a compile time.
- In which the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior.
- It also allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline. Although a dynamically scheduled processor cannot change the data flow, it tries to avoid stalling when dependences, which could generate hazards, are present.

Dynamic Scheduling

- A major limitation of the simple pipelining techniques is that they all use in-order instruction issue and execution: Instructions are issued in program order and if an instruction is stalled in the pipeline, no later instructions can proceed.
- Thus, if there is dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall. If there are multiple functional units, these units could lie idle.
- If instruction j depends on a long-running instruction i, currently in execution in the pipeline, then all instructions after j must be stalled until i is finished and j can execute. For example, consider this code:

```
DIV.D F0, F2,F4 ADD.D F10,F0,F8 SUB.D F12,F8,F14
```

- Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline.

- Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that exactly those exceptions that would arise if the program were executed in strict program order actually do arise.
- Imprecise exceptions can occur because of two possibilities:
- The pipeline may have already completed instructions that are later in program order than the instruction causing the exception, and
- **2.** The pipeline may have not yet completed some instructions that are earlier in program order than the instruction causing the exception.
- To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages:
- 1 **Issue**—Decode instructions, check for structural hazards.
- 2 **Read operands**—Wait until no data hazards, and then read operands.
- In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in- order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order.

- Score-boarding is a technique for allowing instructions to execute out-of-order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard, which developed this capability.
- We focus on a more sophisticated technique, called Tomasulo's algorithm that has several major enhancements over score boarding.

Tomasulo's Approach

Dynamic Scheduling Using Tomasulo's Approach

This scheme was invented by RobertTomasulo, and was first used in the IBM 360/91. it uses register renaming to eliminate output and anti-dependencies, i.e. WAW and WAR hazards.

Output and anti-dependencies are just name dependencies; there is no actual data dependence.

Tomasulo's algorithm implements register renaming through the use of what are called reservation stations.

- Reservation stations are buffers which fetch and store instruction operands as soon as they are available In addition; pending instructions designate the reservation station that will provide their input.

- Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register.
- As instructions are issued, the register specifies for pending operands are renamed to the names of the reservation station, which provides register renaming.

- The basic structure of a Tomasulo-based MIPS processor, including both the floating-point unit and the load/store unit.
- Instructions are sent from the instruction unit into the instruction queue from which they are issued in FIFO order.

- The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards.
- Load buffers have three functions: hold the components of the effective address until it is computed, track outstanding loads that are waiting on the memory, and hold the results of completed loads that are waiting for the CDB.

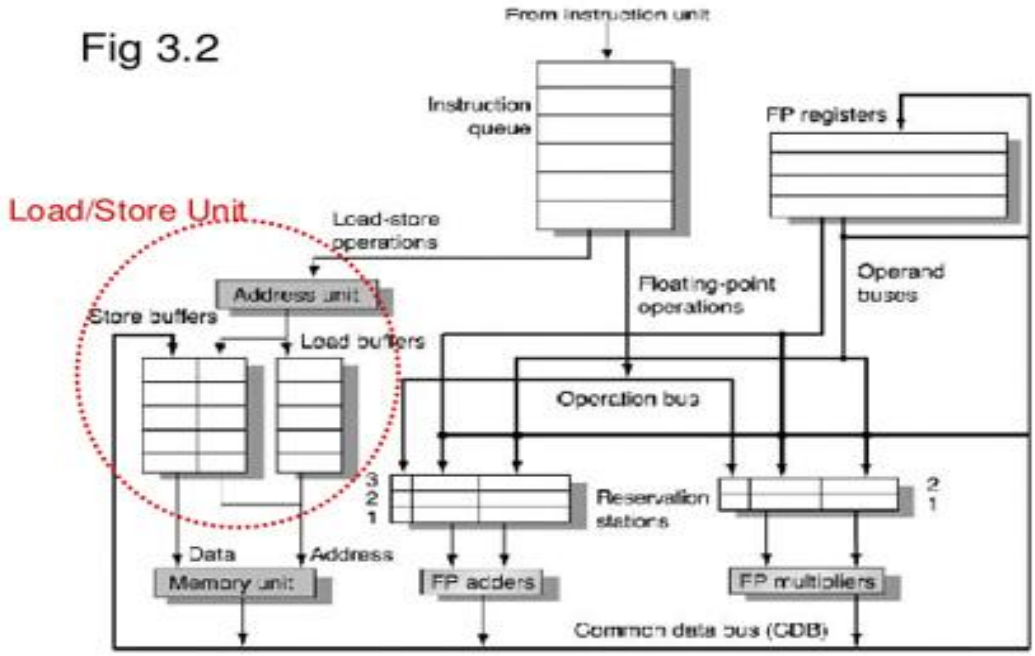
- Similarly, store buffers have three functions: hold the components of the effective address until it is computed, hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and hold the address and value to store until the memory unit is available.

- All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers.

- The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

MIPS floating point unit using Tomasulo's algorithm

Fig 3.2



There are only three steps in Tomasulo's Approach:

- **Issue**—Get the next instruction from the head of the instruction queue.
- If there is a matching reservation station that is empty, issue the instruction to the station with the operand values (renames registers)

- **Execute (EX)**— When all the operands are available, place into the corresponding reservation stations for execution. If operands are not yet available, monitor the common data bus (CDB) while waiting for it to be computed.

- **Write result (WB)—**When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. Stores also write data to memory during this step: When both the address and data value are available, they are sent to the memory unit and the store completes.

- Each reservation station has six fields:

  - **Op**—The operation to perform on source operands S1 and S2.

- **Qj, Qk**—The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.

  - **Vj, Vk**—The value of the source operands. Note that only one of the V field or the Q field is valid for each operand. For loads, the Vk field is used to the offset from the instruction.

  - **A**–used to hold information for the memory address calculation for a load or store.

  - **Busy**—Indicates that this reservation station and its accompanying functional unit are occupied.

Reduce Branch Costs with Dynamic Hardware Prediction

- Basic Branch Prediction and Branch-Prediction Buffers

- The simplest dynamic branch-prediction scheme is a branch-prediction buffer or branch history table.

- A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction.

- The memory contains a bit that says whether the branch was recently taken or not. if the prediction is correct—it may have been put there by another branch that has the same low-order address bits.

- The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.

- The performance of the buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches.

- This simple one-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken.

- The two bits are used to encode the four states in the system. In a counter implementation, the counters are incremented when a branch is taken and decremented when it is not taken; the counters saturate at 00 or 11.

- One complication of the two-bit scheme is that it updates the prediction bits more often than a one-bit predictor, which only updates the prediction bit on a mispredict.

- Since we typically read the prediction bits on every cycle, a two-bit predictor will typically need both a read and a write access port.

- The two-bit scheme is actually a specialization of a more general scheme that has an n-bit saturating counter for each entry in the prediction buffer. With an n-bit counter, the counter can take on values between 0 and 2 maximum value (2 To exploit more ILP, the accuracy of our branch prediction becomes critical, this problem in two ways: by increasing the size of the buffer and by increasing the accuracy of the scheme we use for each prediction.

Correlating Branch Predictors:

- These two-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch.
- It may be possible to improve the prediction accuracy if we also n – 1: when the counter is greater than or equal to one half of its n–1), the branch is predicted as taken; otherwise, it is predicted untaken.
- Look at the recent behavior of other branches rather than just the branch we are trying to predict.
- Consider a small code fragment from the SPEC92 benchmark

```
if (aa==2)
aa=0;
if (bb==2)
bb=0;
if (aa!=bb) {
```

- Here is the MIPS code that we would typically generate for this code fragment assuming that aa and bb are assigned to registers R1 and R2:

```
DSUBUI R3,R1,#2

BNEZ R3,L1 ;branch b1 (aa!=2)
```

```
    DADD R1,R0,R0 ;aa=0
    L1: DSUBUI R3,R2,#2
    BNEZ R3,L2 ;branch b2(bb!=2)
    DADD R2,R0,R0 ; bb=0 L2: DSUBU R3,R1,R2 ;R3=aa-bb
BEQZ R3,L3 ;branch b3 (aa==bb)
```

- Let's label these branches b1, b2, and b3. The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2.
- Clearly, if branches b1 and b2 are both not taken (i.e., the if conditions both evaluate to true and aa and bb are both assigned 0), then b3 will be taken, since aa and bb are clearly equal.
- A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior.

- Branch predictors that use the behavior of other branches to make a prediction are called correlating predictors or two-level predictors. Tournament Predictors: Adaptively Combining Local and Global Predictors

- The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor using only local information failed on some important branches and that by adding global information, the performance could be improved.

- Tournament predictors take this insight to the next level, by using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector.

Hardware speculation

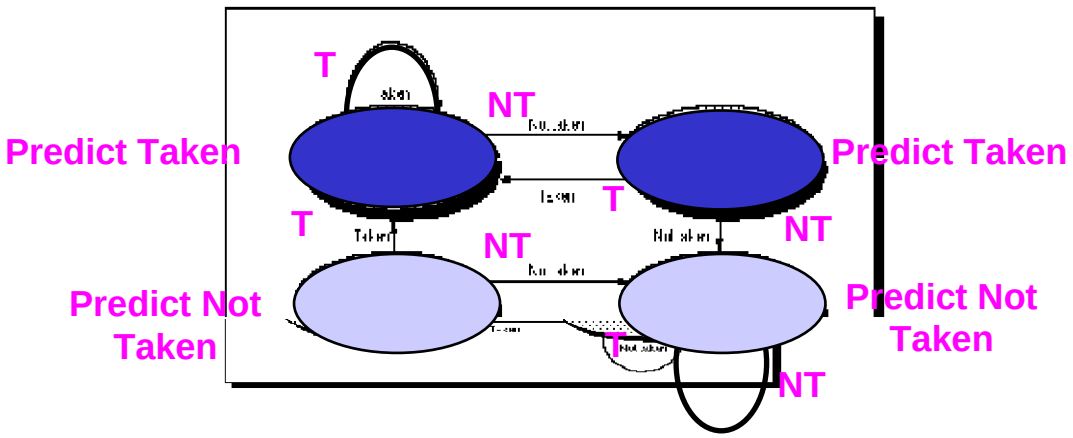
- Hardware-based speculation combines three key ideas: dynamic branch prediction to choose which instructions to execute, speculation to allow the execution of instructions before the control dependences are resolved and dynamic scheduling to deal with the scheduling of different combinations of basic blocks.

- Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions. This method of executing programs is essentially a data-flow execution: operations execute as soon as their operands are available.

- The approach is implemented in a number of processors (PowerPC 603/604/G3/G4, MIPS R10000/R12000, Intel Pentium II/III/ 4, Alpha 21264, and AMD K5/K6/Athlon), is to implement speculative execution based on Tomasulo's algorithm.

- The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit in order and to prevent any irrevocable action until an instruction commits.

    - In the simple single-issue five-stage pipeline we could ensure that instructions committed in order, and only after any exceptions for that instruction had been detected, simply by moving writes to the end of the pipeline.



Limitations of ILP

The Hardware Model

- An ideal processor is one where all artificial constraints on ILP are removed. The only limits on ILP in such a processor are those imposed by the actual data flows either through registers or memory.

- The assumptions made for an ideal or perfect processor are as follows:

- **Register renaming**—There are an infinite number of virtual registers available and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.

- **Branch prediction**—Branch prediction is perfect. All conditional branches are predicted exactly.

- **Jump prediction**—All jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.

- **Memory-address alias analysis**—All memory addresses are known exactly

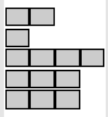
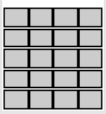
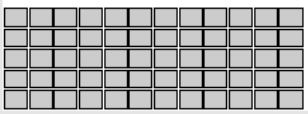
UNIT II

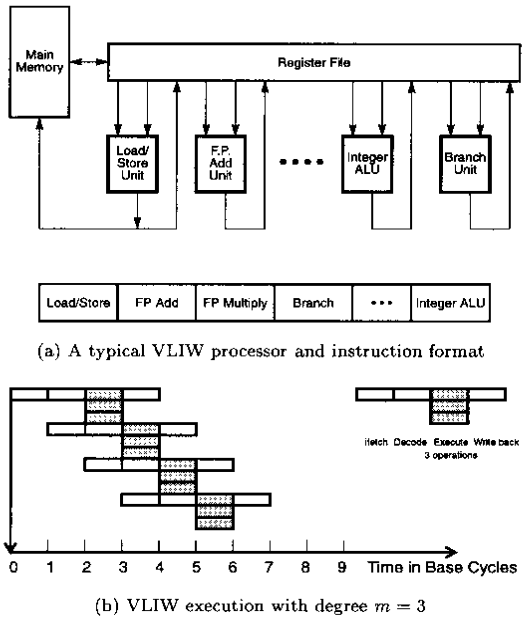
MULTIPLE ISSUE PROCESSORS
VLIW & EPIC – Advanced compiler support – Hardware support for exposing parallelism – Hardware versus software speculation mechanisms – IA 64 and Itanium processors – Limits on ILP.

The VLIW Architecture

- A typical VLIW (very long instruction word) machine has instruction words hundreds of bits in length.
- Multiple functional units are used concurrently in a VLIW processor.
- All functional units share the use of a common large register file.

Comparison: CISC, RISC, VLIW

ARCHITECTURE CHARACTERISTIC	CISC	RISC	VLIW
INSTRUCTION SIZE	Varies	One size, usually 32 bits	One size
INSTRUCTION FO RMAT	Field placement varies	Regular, consistent placement of fields	Regular, consistent placement of fields
INSTRUCTION SEMANTICS	Varies from simple to complex; possibly many dependent operations per instruction	Almost always one simple operation	Many simple, independent operations
REGISTERS	Few, sometimes special	Many, general-purpose	Many, general-purpose
MEMORY REFE RENCES	Bundled with operations in many different types of instructions	Not bundled with operations, i.e., load/store architecture	Not bundled with operations, i.e., load/store architecture
HARDWARE DESIGN FOCUS	Exploit microcoded implementations	Exploit implementations with one pipeline and & no microcode	Exploit implementations with multiple pipelines, no microcode & no complex dispatch logic
PICTURE OF FIVE TYPICAL INSTRU CTIONS = 1 BYTE			



(a) A typical VLIW processor and instruction format

(b) VLIW execution with degree $m = 3$

Figure 4.14 The architecture of a very long instruction word (VLIW) processor and its pipeline operations. (Courtesy of Multiflow Computer, Inc., 1987)

Advantages of VLIW

- C compiler prepares fixed packets of multiple operations that give the full "plan of execution"
  - dependencies are determined by compiler and used to schedule according to function unit latencies
  - function units are assigned by compiler and correspond to the position within the instruction packet ("slotting")
  - compiler produces fully-scheduled, hazard-free code => hardware doesn't have to "rediscover" dependencies or schedule

Disadvantages of VLIW

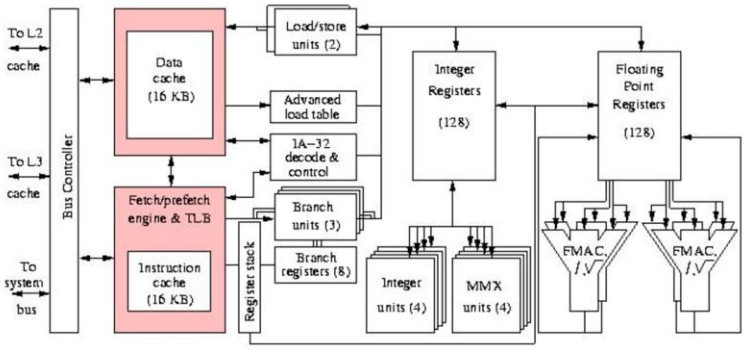
- Compatibility across implementations is a major problem
- VLIW code won't run properly with different number of function units or different latencies
- unscheduled events (e.g., cache miss) stall entire processor
- Code density is another problem
- low slot utilization (mostly nops)
- reduce nops by compression ("flexible VLIW", "variable-length VLIW")

EPIC an Introduction

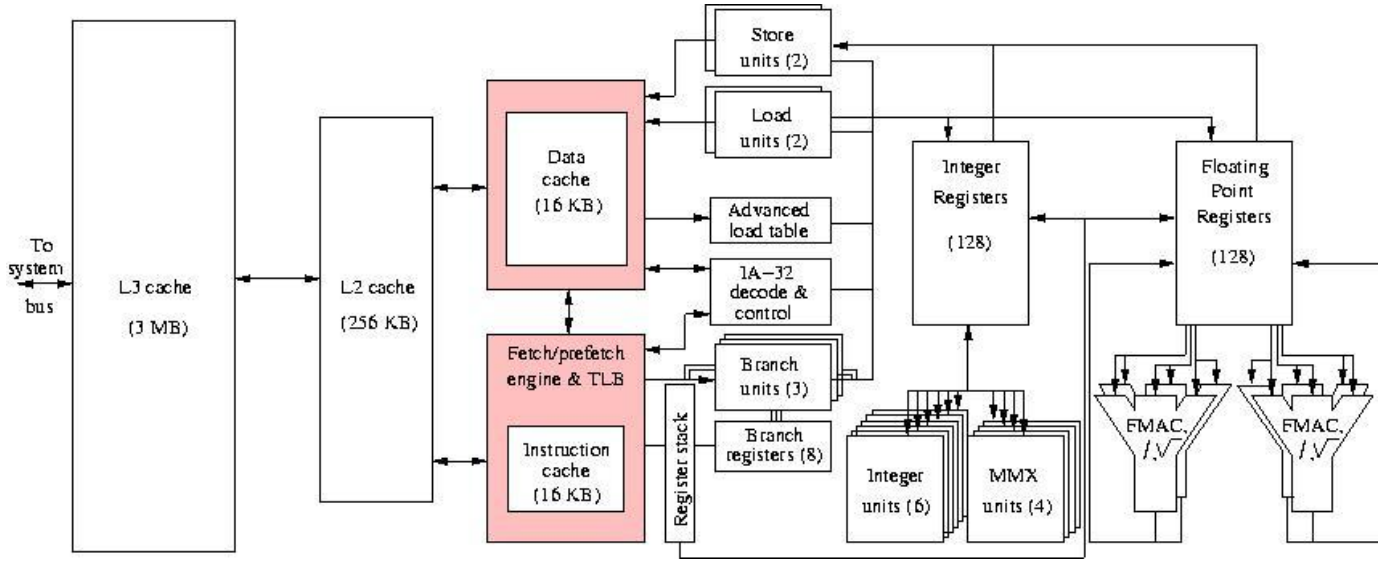
- EPIC
- –Explicitly Parallel Instruction Computing

- –instruction Level Parallelism (ILP) is identified to hardware by
- the compiler.
- Particular EPIC architecture we cover today: IA64
- 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: IA64 - Intel Itanium



Block diagram of Intel Intanium.



I

- 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 1.66 GHz on Montvale
- Itanium 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

Compiler Support For ILP

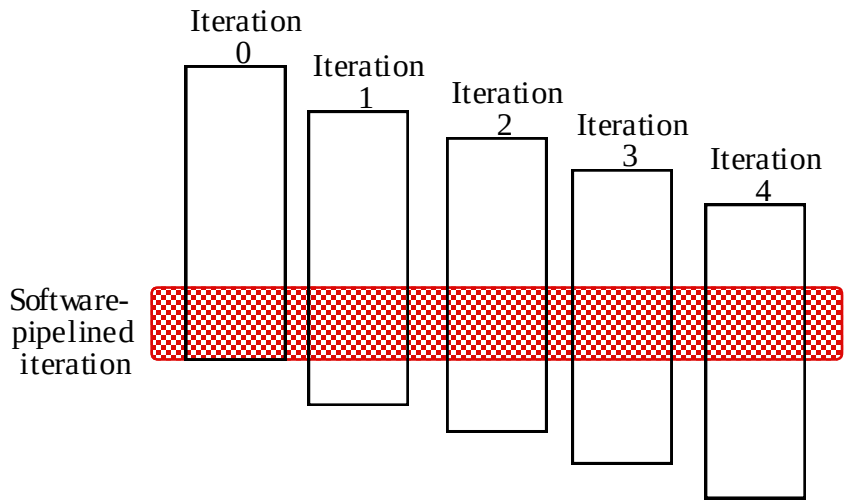
- Produce good scheduling of code.
- Determine which loops might contain parallelism.
- Eliminate name dependencies.
- Compilers must be REALLY smart to figure out aliases -- pointers in C are a real problem.

Techniques lead to

- Symbolic Loop Unrolling
- Critical Path Scheduling

Software Pipelining

- Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (Tomasulo in SW)



SW Pipelining Example

Before: Unrolled 3 times

```
1    LD    F0,0(R1)
2    ADDD  F4,F0,F2
3    SD    0(R1),F4
4    LD    F6,-8(R1)
5    ADDD  F8,F6,F2
6    SD    -8(R1),F8
7    LD    F10,-16(R1)
```

```
8     ADDD F12,F10,F2
9     SD   -16(R1),F12
10    SUBI R1,R1,#24
11    BNEZ R1,LOOP
```

Hardware Support for Exposing Parallelism

- Techniques such as loop unrolling, software pipelining, and trace scheduling can be used to increase the amount of parallelism available when the behavior of branches is fairly predictable at compile time.
- When the behavior of branches is not well known, compiler techniques alone may not be able to uncover much ILP.
- In such cases, the control dependences may severely limit the amount of parallelism that can be exploited. To overcome these problems, an architect can extend the instruction set to include conditional or predicated instructions.
- Such instructions can be used to eliminate branches, converting control dependence into a data dependence and potentially improving performance.
- Such approaches are useful with either the hardware-intensive schemes in or the software-intensive approaches discussed in this appendix, since in both cases predication can be used to eliminate branches.
- The concept behind conditional instructions is quite simple: An instruction refers to a condition, which is evaluated as part of the instruction execution.
- If the condition is true, the instruction is executed normally; if the condition is false, the execution continues as if the instruction were a no-op. Much newer architecture include some form of conditional instructions.
- The most common example of such an instruction is conditional move, which moves a value from one register to another if the condition is true.
- Such an instruction can be used to completely eliminate a branch in simple code sequences.

Example

Consider the following code:

- Assuming that registers R1, R2, and R3 hold the values of A, S, and T, respectively, show the code for this statement with the branch and with the conditional move.

Answer

- The straightforward code using a branch for this statement is (remember that we are assuming normal rather than delayed branches)
- Using a conditional move that performs the move only if the third operand is equal to zero, we can implement this statement in one instruction:
- The conditional instruction allows us to convert the control dependence present in the branch-based code sequence to a data dependence. (This transformation is also used for vector computers, where it is called if conversion.)
- For a pipelined processor, this moves the place where the dependence must be resolved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline, where the register write occurs.
- One obvious use for conditional move is to implement the absolute value function: A = abs (B), which is implemented as if (B<0) {A=−B;} else {A=B;}.
- This if statement can be implemented as a pair of conditional moves, or as one unconditional move (A=B) and one conditional move (A=−B).
- In the example above or in the compilation of absolute value, conditional moves are used to change a control dependence into a data dependence.
- This enables us to eliminate the branch and possibly improve the pipeline behavior. As issue rates increase, designers are faced with one of two choices: execute multiple branches per clock cycle or find a method to eliminate branches to avoid this requirement.
- Handling multiple branches per clock is complex, since one branch must be control dependent on the other.
- The difficulty of accurately predicting two branch outcomes, updating the prediction tables, and executing the correct sequence has so far caused most designers to avoid processors that execute multiple branches per clock.
- Conditional moves and predicated instructions provide a way of reducing the branch pressure. In addition, a conditional move can often eliminate a branch that is hard to predict, increasing the potential gain.
- Conditional moves are the simplest form of conditional or predicated instructions and, although useful for short sequences, have limitations.
- In particular, using conditional move to eliminate branches that guard the execution of large blocks of code can be inefficient, since many conditional moves may need to be introduced.

- To remedy the inefficiency of using conditional moves, some architectures support full predication, whereby the execution of all instructions is controlled by a predicate.
- When the predicate is false, the instruction becomes a no-op. Full predication allows us to simply convert large blocks of code that are branch dependent.
- For example, an if-then-else statement within a loop can be entirely converted to predicated execution, so that the code in the then case executes only if the value of the condition is true and the code in the else case executes only if the value of the condition is false.
- Predication is particularly valuable with global code scheduling, since it can eliminate nonloop branches, which significantly complicate instruction scheduling.
- Predicated instructions can also be used to speculatively move an instruction that is time critical, but may cause an exception if moved before a guarding branch. Although it is possible to do this with conditional move, it is more costly.

Example

- Here is a code sequence for a two-issue superscalar that can issue a combination of one memory reference and one ALU operation, or a branch by itself, every cycle:
- This sequence wastes a memory operation slot in the second cycle and will incur a data dependence stall if the branch is not taken, since the second LW after the branch depends on the prior load. Show how the code can be improved using a predicated form of LW.

Answer

- Call the predicated version load word LWC and assume the load occurs unless the third operand is 0. The LW immediately following the branch can be converted to an LWC and moved up to the second issue slot:
- This improves the execution time by several cycles since it eliminates one instruction issue slot and reduces the pipeline stall for the last instruction in the sequence. Of course, if the compiler mispredicted the branch, the predicated instruction will have no effect and will not improve the running time. This is why the transformation is speculative.
- If the sequence following the branch were short, the entire block of code might be converted to predicated execution and the branch eliminated.
- When we convert an entire code segment to predicated execution or speculatively move an instruction and make it predicted, we remove a control dependence. Correct code generation and the conditional execution of predicated instructions ensure that we maintain the data flow enforced

by the branch. To ensure that the exception behavior is also maintained, a predicated instruction must not generate an exception if the predicate is false.

- The property of not causing exceptions is quite critical, as the previous example shows: If register R10 contains zero, the instruction LW R8,0(R10) executed unconditionally is likely to cause a protection exception, and this exception should not occur. Of course, if the condition is satisfied (i.e., R10 is not zero), the LW may still cause a legal and resumable exception (e.g., a page fault), and the hardware must take the exception when it knows that the controlling condition is true.
- The major complication in implementing predicated instructions is deciding when to annul an instruction. Predicated instructions may either be annulled during instruction issue or later in the pipeline before they commit any results or raise an exception. Each choice has a disadvantage.
- If predicated instructions are annulled early in the pipeline, the value of the controlling condition must be known early to prevent a stall for a data hazard. Since data-dependent branch conditions, which tend to be less predictable, are candidates for conversion to predicated execution, this choice can lead to more pipeline stalls.
- Because of this potential for data hazard stalls, no design with predicated execution (or conditional move) annuls instructions early. Instead, all existing processors annul instructions later in the pipeline, which means that annulled instructions will consume functional unit resources and potentially have a negative impact on performance.
- A variety of other pipeline implementation techniques, such as forwarding, interact with predicated instructions, further complicating the implementation.

Hardware Support for Compiler Speculation

Compiler needs to move instructions before branch, possibly before condition

- Requirements:
  - Instructions that can be moved without disrupting data flow
  - Exceptions that can be ignored until outcome is known
  - Ability to speculatively access memory with potential address conflicts
- Four methods:
  - Hardware and OS cooperate to ignore exceptions for speculative instructions
  - Speculative instructions *never* raise exceptions; explicit checks must be made
  - Poison bits used to mark registers with invalid results; use causes exception
  - Speculative results are buffered until certain

Hardware versus software speculation mechanisms

A number of trade-offs and limitations

- Disambiguating memory references is hard for a compiler
- Hardware branch prediction is usually better
- Precise exceptions easier in hardware
- Hardware does not require "housekeeping" code
- Compilers can "look" further
- Hardware techniques are more portable

- Major disadvantage of hardware: *complexity!*
- Some architectures combine hardware and software approaches.
-

IA 64 and Itanium processors

- IA-64
- RISC-style
  - Register-register
  - Emphasis on software-based optimizations

Features

- 128 × 65-bit integer registers
- 128 × 82-bit FP registers
- 64 predicate registers; 8 branch registers

Integer registers

- Use windowing mechanism
- 0–31 always visible
- Remainder arranged in overlapping windows
- Local and out areas (variable size)
- Hardware for over-/underflow
- Int and FP registers support register rotation
- Supports software pipelining

Instruction Format and VLIW

- Compiler schedules parallel instructions; flags dependences
- Instruction group
  - Sequence of (register) independent instructions
  - Compiler marks boundaries between groups (*stop*)
- Bundle
  - 128-bits: 5-bit template + 3 × 41-bit instructions

Instruction Bundle

- Template specifies *stops* and *execution unit*
  - I-unit (int + special — multimedia, etc.)
  - M-unit (int + memory access)
  - F-unit (FP)
  - B-unit (branches)
  - L+X (extended instructions)

Example

```
for (int k = 0; k < 1000; k++)
 { x[k]   = x[k] + s;
 }
```

Unrolled seven times

Optimized for size:

9 bundles; 15% nops

21 cycles (3 per calculation)

Optimized for performance:

11 bundles; 30% nops

12 cycles (1.7 per calculation)

Instructions

- 41-bits long
  - 4-bit opcode (+ template bits)
  - 6-bit predicate register specifier

- Predication
  - Almost all instructions can be predicated
    - Branch is jump with predicate check!
  - Complex comparisons set two predicate registers

Speculation

- Exceptions can be deferred
  - Uses poison bits (65-bit registers)
  - Nonspeculative and **chk** instructions raise exception
- Speculative loads
  - Called advanced load (**ld.a**)
  - Stores check addresses

Itanium

- First implementation of IA-64
- Issues up to six instructions per cycle (two bundles)
- Nine functional units
- 2 × I, 2 × M, 3 × B, 2 × F
- 10-stage pipeline
- Multilevel dynamic branch predictor
- Complex hardware with many features of dynamically scheduled pipelines!
- Branch prediction
- Register renaming
- Scoreboarding
- Deep pipeline etc.

Limits to ILP

- Initial HW Model here; MIPS compilers.
- Assumptions for ideal/perfect machine to start:
- Register renaming – infinite virtual registers
  all register WAW & WAR hazards are avoided
- Branch prediction – perfect; no mispredictions

- Jump prediction – all jumps perfectly predicted (returns, case statements) no control dependencies; perfect speculation & an unbounded buffer of instructions available

4. Memory-address alias analysis – addresses known & a load can be moved before a store provided addresses not equal; 1&4 eliminates all but RAW

Also: perfect caches; 1 cycle latency for all instructions (FP *,/); unlimited instructions issued/clock cycle;

Limits to ILP HW Model comparison

	Model	Power 5
Instructions Issued per clock	Infinite	4
Instruction Window Size	Infinite	200
Renaming Registers	Infinite	88 integer + 88 Fl. Pt.
Branch Prediction	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias Analysis	Perfect	??

- Doubling issue rates above today's 3-6 instructions per clock, say to 6 to 12 instructions, probably requires a processor to
- issue 3 or 4 data memory accesses per cycle,
- resolve 2 or 3 branches per cycle,
- rename and access more than 20 registers per cycle, and
- Fetch 12 to 24 instructions per cycle.
- The complexities of implementing these capabilities is likely to mean sacrifices in the maximum clock rate
- E.g., widest issue processor is the Itanium 2, but it also has the slowest clock rate, despite the fact that it consumes the most power!
- Most techniques for increasing performance increase power consumption
- Multiple issue processors techniques all are energy inefficient:
- Issuing multiple instructions incurs some overhead in logic that grows faster than the issue rate grows
- Growing gap between peak issue rates and sustained performance

Number of transistors switching = f(peak issue rate), and performance = f(sustained rate), growing gap between peak and sustained performance

$\Rightarrow$ increasing energy per unit of performance

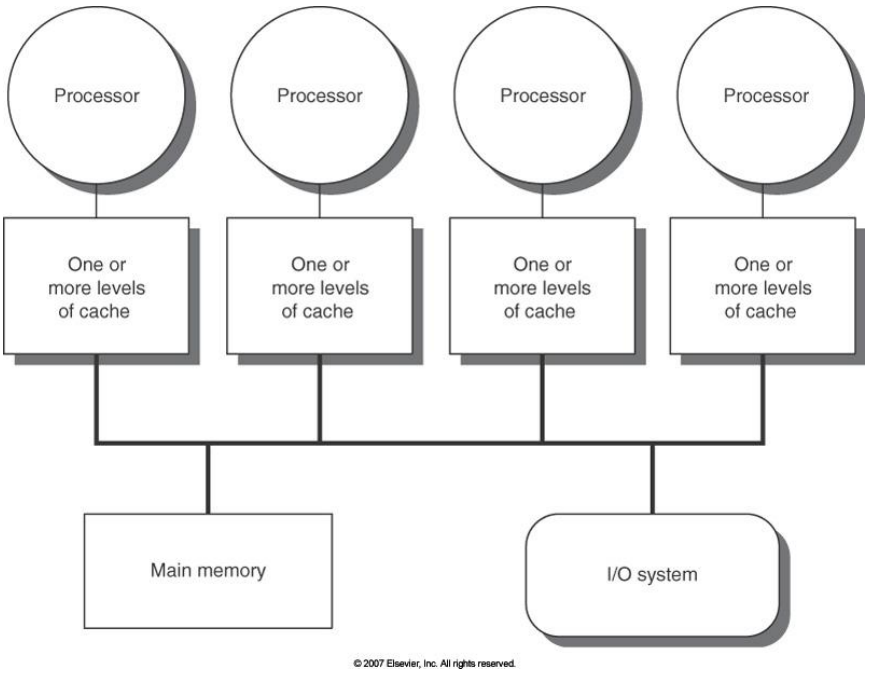
UNIT III

UNIT III MULTIPROCESSORS AND THREAD LEVEL PARALLELISM

Symmetric and distributed shared memory architectures – Performance issues – Synchronization – Models of memory consistency – Introduction to Multithreading.

Symmetric Shared Memory Architectures

- The Symmetric Shared Memory Architecture consists of several processors with a single physical memory shared by all processors through a shared bus which is shown below.



- Small-scale shared-memory machines usually support the caching of both shared and private data. Private data is used by a single processor, while shared data is used by multiple processors; essentially providing communication among the processors through reads and writes of the shared data.

- When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required.

- Since no other processor uses the data, the program behavior is identical to that in a uniprocessor.

Cache Coherence in Multiprocessors

- Introduction of caches caused a coherence problem for I/O operations;

- The same problem exists in the case of multiprocessors, because the view of memory held by two different processors is through their individual caches.

- The problem and shows how two different processors can have two different values for the same location. This difficulty s generally referred to as the 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

FIGURE 6.7 The cache-coherence problem for a single memory location (X), read and written by two processors (A and B).

- We initially assume that neither cache contains the variable and that X has the value 1. We also assume a write-through cache; a write-back cache adds some additional but similar complications.
- After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 1!
- Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item.
- This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs.
- The first aspect, called coherence, defines what values can be returned by a read. The second aspect, called consistency, determines when a written value will be returned by a read. Let's look at coherence first.

A memory system is coherent if

- A read by a processor, P, to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
- A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
- Writes to the same location are serialized: that is, two writes to the same location by any two processors are seen in the same order by all processors.

- For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

- Coherence and consistency are complementary: Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations.

Basic Schemes for Enforcing Coherence

- Coherent caches provide migration, since a data item can be moved to a local cache and used there in a transparent fashion.
- This migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.

- Coherent caches also provide replication for shared data that is being simultaneously read, since the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item.

- The protocols to maintain coherence for multiple processors are called cache- coherence protocols. There are two classes of protocols, which use different techniques to track the sharing status, in use:

- **Directory based**—The sharing status of a block of physical memory is kept in just one location, called the **directory;** we focus on this approach in section 6.5, when we discuss scalable shared-memory architecture.

- **Snooping**—Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept.

- The caches are usually on a shared-memory us, and all cache controllers monitor or snoop on the bus to determine whether or not they have a copy of a block that is requested on the bus.

Snooping Protocols

- The method which ensure that a processor has exclusive access to a data item before it writes that item. this style of protocol is called a **write invalidate protocol** because it invalidates other copies on a write.

- It is by far the most common protocol, both for snooping and for directory schemes. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.
- Since the write requires exclusive access, any copy held by the reading processor must be invalidated hence the protocol name). Thus, when the read occurs, it misses in the cache and is

forced to fetch a new copy of the data.

- For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write simultaneously.
- If two processor attempt to write the same data simultaneously, one of them wins the race, causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore, this protocol enforces write serialization.

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	0CPU 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.

- The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written.
- This type of protocol is called a write update or write broadcast protocol. Figure shows an example of a write update protocol in operation.
- In the decade since these protocols were developed, invalidate has emerged as the winner for the vast majority of designs.

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	0CPU B reads X	Cache miss for X
	0	0	0	
CPU A writes a 1	Write broadcast	1	1	1

to X	of X			
CPU B reads X		1	1	1

FIGURE 6.9 An example of a write update or broadcast protocol working on a snooping bus for a single cache block (X) with write-back caches.

- The performance differences between write update and write invalidate protocols arise from three characteristics:
- Multiple writes to the same word with no intervening reads require multiple write broadcasts in an update protocol, but only one initial invalidation in a write invalidate protocol.

- With multiword cache blocks, each word written in a cache block requires a write broadcast in an update protocol, although only the first write to any word in the block needs to generate an invalidate in an invalidation protocol.
- An invalidation protocol works on cache blocks, while an update protocol must work on individual words (or bytes, when bytes are written).

- It is possible to try to merge writes in a write broadcast scheme.

- The delay between writing a word in one processor and reading the written value in another processor is usually less in a write update scheme, since the written data are immediately updated in the reader's cache

Basic Implementation Techniques

- The serialization of access enforced by the bus also forces serialization of writes, since when two processors compete to write to the same location, one must obtain bus access before the other.
- The first processor to obtain bus access will cause the e other processor's copy to be invalidated, causing writes to be strictly serialized.
- One implication of this scheme is that a write to a shared data item cannot complete until it obtains bus access.
- For a write-back cache, however, the problem of finding the most recent data value is harder, since the most recent value of a data item can be in a cache rather than in memory.
- write-back caches can use the same snooping scheme both for caches misses and for writes: Each processor snoops every address placed on the bus.

- If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted.
- Since Write-back caches generate lower requirements for memory bandwidth, they are greatly preferable in a multiprocessor, despite the slight increase in complexity. Therefore, we focus on implementation with write-back caches.

- The normal cache tags can be used to implement the process of snooping, and the valid bit for each block makes invalidation easy to implement.
- Read misses, whether generated by invalidation or by some other event, are also straightforward since they simply rely on the snooping capability.
- For writes we'd like to know whether any other copies of the block are cached, because, if there are no other cached copies, then the write need not be placed on the bus in a write-back cache. Not sending the write reduces both the time taken by the write and the required bandwidth.

Distributed Shared-Memory Architectures.

There are several disadvantages in Symmetric Shared Memory architectures.

- First, compiler mechanisms for transparent software cache coherence are very limited.
- Second, without cache coherence, the multiprocessor loses the advantage of being able to fetch and use multiple words in a single cache block for close to the cost of fetching one word.

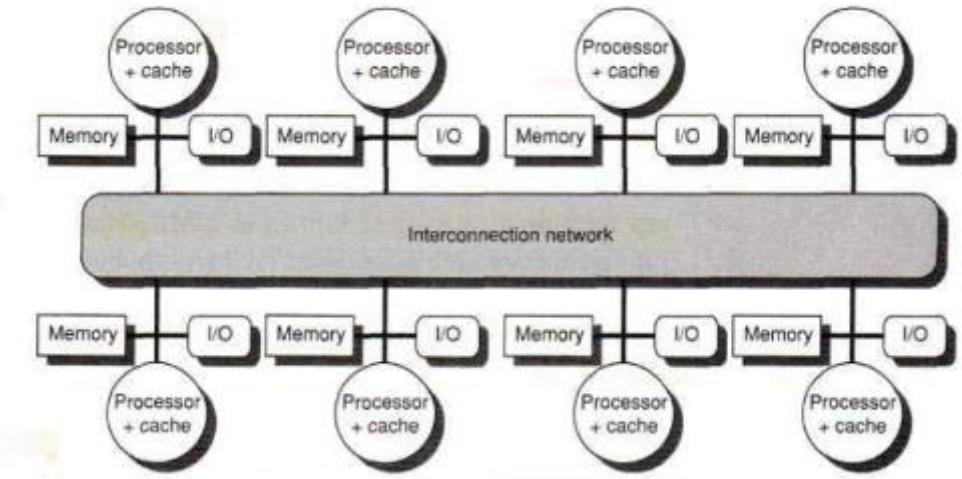
- Third, mechanisms for tolerating latency such as prefetch are more useful when they can fetch multiple words, such as a cache block, and where the fetched data remain coherent; we will examine this advantage in more detail later.

- These disadvantages are magnified by the large latency of access to remote memory versus a local cache. For these reasons, cache coherence is an accepted requirement in small-scale multiprocessors.

- For larger-scale architectures, there are new challenges to extending the cache- coherent shared-memory model.
- Although the bus can certainly be replaced with a more scalable interconnection network and we could certainly distribute the memory so that the memory bandwidth could also be scaled, the lack of scalability of the snooping coherence scheme needs to be addressed is known as Distributed Shared Memory architecture.

- The first coherence protocol is known as a directory protocol. A directory keeps the state of every block that may be cached. Information in the directory includes which caches have copies of the block, whether it is dirty, and so on.

- To prevent the directory from becoming the bottleneck, directory entries can be distributed along with the memory, so that different directory accesses can go to different locations, just as different memory requests go to different memories.
- A distributed directory retains the characteristic that the sharing status of a block is always in a single known location. This property is what allows the coherence protocol to avoid broadcast. Figure 6.27 shows how our distributed-memory multiprocessor looks with the directories added to each node.



A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor.

Directory-Based Cache-Coherence Protocols: The Basics

There are two primary operations that a directory protocol must implement:

- Handling a read miss and handling a write to a shared, clean cache block. (Handling a write miss to a shared block is a simple combination of these two.)

- To implement these operations, a directory must track the state of each cache block. In a simple protocol, these states could be the following:

- **Shared**—One 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
- **Exclusiv***e*—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, since they will need to be invalidated on a write.
- The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block.
- We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

- A catalog of the message types that may be sent between the processors and the directories. Figure shows the type of messages sent among nodes. The **local** node is the node where a request originates.

- The **home** node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known.

- For example, the high-order bits may provide the node number, while the low-order bits provide the offset within the memory on that node.
- The local node may also be the home node. The directory must be accessed when the home node is the local node, since copies may exist in yet a third node, called a remote node.

- A **remote** node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared.
- A remote node may be the same as either the local node or the home node. In such cases, the basic protocol does not change, but interprocessor messages may be replaced with intraprocessor messages.

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.

Synchronization and various Hardware Primitives

Synchronization

- Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions.
- The efficient spin locks can be built using a simple hardware synchronization instruction and the coherence mechanism.

Basic Hardware Primitives

- The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location.
- Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases.
- There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically.

- These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers.
- One typical operation for building synchronization operations is the atomic exchange, which interchanges a value in a register for a value in memory.
- Use this to build a basic synchronization operation, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and a 1 is used to indicate that the lock is unavailable.

- A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock.
- The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise. In the latter case, the value is also changed to be 1, preventing any competing exchange from also retrieving a 0.

- There are a number of other atomic primitives that can be used to implement synchronization. They all have the key property that they read and update a memory value in such a manner that we can tell whether or not the two operations executed atomically.
- One operation, present in many older multiprocessors, is *test-and-set*, which tests a value and sets it if the value passes the test. For example, we could define an operation that tested for 0 and set the value to 1, which can be used in a fashion similar to how we used atomic exchange.

- Another atomic synchronization primitive is fetch-and-increment*:* it returns the value of a memory location and atomically increments it.
- By using the value 0 to indicate that the synchronization variable is unclaimed, we can use fetch-and-increment, just as we used exchange. There are other uses of operations like fetch-and-increment.

Implementing Locks Using Coherence

- We can use the coherence mechanisms of a multiprocessor to implement spin locks: locks that a processor continuously tries to acquire, spinning around a loop until it succeeds.
- Spin locks are used when a programmer expects the lock to be held for a very short amount of time and when she wants the process of locking to be low latency when the lock is available.

- Because spin locks tie up the processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances.
- The simplest implementation, which we would use if there were no cache coherence, would keep the lock variables in memory.

- A processor could continually try to acquire the lock using an atomic operation, say exchange, and test whether the exchange returned the lock as free.
- To release the lock, the processor simply stores the value 0 to the lock. Here is the code sequence to lock a spin lock whose address is in R1 using an atomic exchange:

```
DADDUI R2,R0,#1
lockit: EXCH    R2,0(R1)    ; atomic exchange
BNEZ    R2,lockit    ; already locked?
```

- If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently.

Caching locks has two advantages.

- First, it allows an implementation where the process of "spinning" (trying to test and acquire the lock in a tight loop) could be done on a local cached copy rather than requiring a global memory access on each attempt to acquire the lock.
- The second advantage comes from the observation that there is often locality in lock accesses: that is, the processor that used the lock last will use it again in the near future. In such cases, the lock value may reside in the cache of that processor, greatly reducing the time to acquire the lock.

Synchronization Performance Challenges

Barrier Synchronization

- One additional common synchronization operation in programs with parallel loops is a barrier. A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes.
- A typical implementation of a barrier can be done with two spin locks: one used to protect a counter that tallies the processes arriving at the barrier and one used to hold the processes until the last process arrives at the barrier.

Synchronization Mechanisms for Larger-Scale Multiprocessors

Software Implementations

- The major difficulty with our spin-lock implementation is the delay due to contention when many processes are spinning on the lock.
- One solution is to artificially delay processes when they fail to acquire the lock.
- The best performance is obtained by increasing the delay exponentially whenever the attempt to acquire the lock fails.
- Figure shows how a spin lock with exponential back-off is implemented. Exponential back-off is a common technique for reducing contention in shared resources, including access to shared networks and buses.
- This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop.
- The result is that if many processes are waiting, the back-off does not affect the processes on their first attempt to acquire the lock. We could also delay that process, but the result would be poorer performance when the lock was in use by only two processes and the first one happened to find it locked.

```
ADDUI   R3,R0,#1    ;R3 = initial delay
lockit: LL  R2,0(R1)    ;load linked
 BNEZ   R2,lockit    ;not available-spin
 DADDUI R2,R2,#1 ;get locked value
 SC     R2,0(R1)    ;store conditional
 BNEZ   R2,gotit    ;branch if store succeeds
 DSLL   R3,R3,#1    ;increase delay by factor of 2
 PAUSE R3           ;delays by value in R3
 J      lockit
          gotit:    use data protected by lock
```

A spin lock with exponential back-off.

- Another technique for implementing locks is to use queuing locks. Queuing locks work by constructing a queue of waiting processors; whenever a processor frees up the lock, it causes the next processor in the queue to attempt access.
- This eliminates contention for a lock when it is freed. We show how queuing locks operate in the next section using a hardware implementation, but software implementations using arrays can achieve most of the same benefits Before we look at hardware primitives,

Hardware Primitives

- In this section we look at two hardware synchronization primitives.
- The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices.
- In both cases we can create hardware primitive where latency is essentially identical to our earlier version, but with much less serialization, leading to better scaling when there is contention.
- The major problem with our original lock implementation is that it introduces a large amount of unneeded contention.
- For example, when the lock is released all processors generate both a read and a write miss, although at most one processor can successfully get the lock in the unlocked state. This sequence happens on each of the 20 lock/unlock sequences.
- It can improve this situation by explicitly handing the lock from one waiting processor to the next. Rather than simply allowing all processors to compete every time the lock is released, we keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes.
- This sort of mechanism has been called a *queuing lock*. Queuing locks can be implemented either in hardware, or in software using an array to keep track of the waiting processes.

Multithreading: Exploiting Thread-Level Parallelism within a Processor

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion.

To permit this sharing, the processor must duplicate the independent state of each thread. For example, a separate copy of the register file, a separate PC, and a separate page table are required for each thread.

There are two main approaches to multithreading.

- **Fine-grained multithreading** switches between threads on each instruction, causing the execution of multiples threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time.
- Switches between threads on each instruction, causing the execution of multiples threads to be
- interleaved
- Usually done in a round-robin fashion, skipping any stalled threads• CPU must be able to switch threads every clock

- Advantage is it can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage is it slows down execution of individual
- since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Sun's Niagara
-
- **Coarse-grained multithreading** was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level two caches misses.

- This change relieves the need to have thread-switching be essentially free and is much less likely to slow the processor down, since instructions from other threads will only be issued, when a thread encounters a costly stall.
- Switches threads only on costly stalls, such as L2 cache misses Used in IBM AS/400

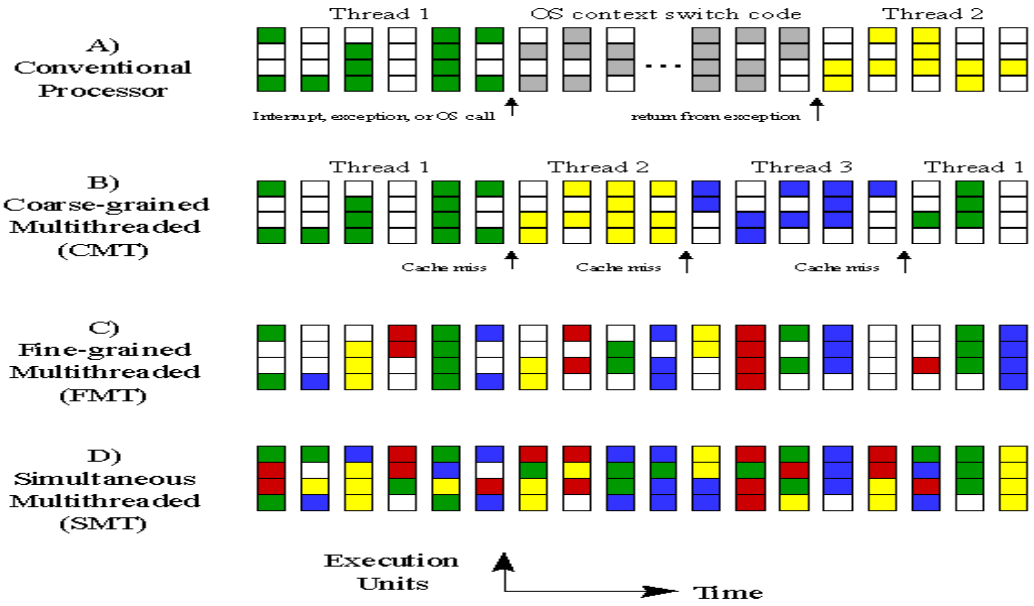
- Advantages Relieves need to have very fast thread-switching
- Doesn't slow down thread, since instructions from other threads issued only when the thread encounters
- a costly stall
- Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
- Since CPU issues instructions from 1 thread,.
- when a stall occurs, the pipeline must be emptied or frozen
- New thread must fill pipeline before instructions can complete Because of start-up overhead, coarse-grained multithreading better at reducing penalty of high cost stalls, where pipeline refill << stall time

Simultaneous Multithreading: Converting Thread-Level Parallelism into Instruction- Level Parallelism: Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple issue, dynamically-scheduled processor to exploit TLP at the same time it exploits ILP.

- The key insight that motivates SMT is that modern multiple- issue processors often have more functional unit parallelism available than a single thread can effectively use. Furthermore, with

register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

- Figure conceptually illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configurations:
- n a superscalar with no multithreading support,
- n a superscalar with coarse-grained multithreading,
- n a superscalar with fine-grained multithreading, and n a superscalar with simultaneous multithreading.



Superscalar without multithreading support, the use of issue slots is limited by a lack of ILP.

- In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor.
- In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle.
- In the SMT case, thread-level parallelism (TLP) and instruction-level parallelism (ILP) are exploited simultaneously; with multiple threads using the issue slots in a single clock cycle.
- Figure greatly simplifies the real operation of these processors it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

Design Challenges in processors

There are a variety of design challenges for an SMT processor, including:

- Dealing with a larger register file needed to hold multiple contexts,
- Maintaining low overhead on the clock cycle, particularly in critical steps such as instruction issue, where more candidate instructions need to be considered, and in instruction completion, where choosing what instructions to commit may be challenging, and
- Ensuring that the cache conflicts generated by the simultaneous execution of multiple threads do not cause significant performance degradation.

In viewing these problems, two observations are important. In many cases, the potential performance overhead due to multithreading is small, and simple choices work well enough. Second, the efficiency of current super-scalars is low enough that there is room for significant improvement, even at the cost of some overhead.

Design Challenges in SMT

- Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on
- single thread performance
- A preferred thread approach sacrifices neither throughput nor single-thread performance.
- Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput,
- when preferred thread stalls
- Larger register file needed to hold multiple contexts
- Not affecting clock cycle time, especially in – Instruction issue - more candidate instructions
- need to be considered
- Instruction completion - choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance

UNIT IV

MEMORY AND I/O

Cache performance – Reducing cache miss penalty and miss rate – Reducing hit time – Main memory and performance – Memory technology. Types of storage devices – Buses – RAID – Reliability, availability and dependability – I/O performance measures – Designing an I/O system.

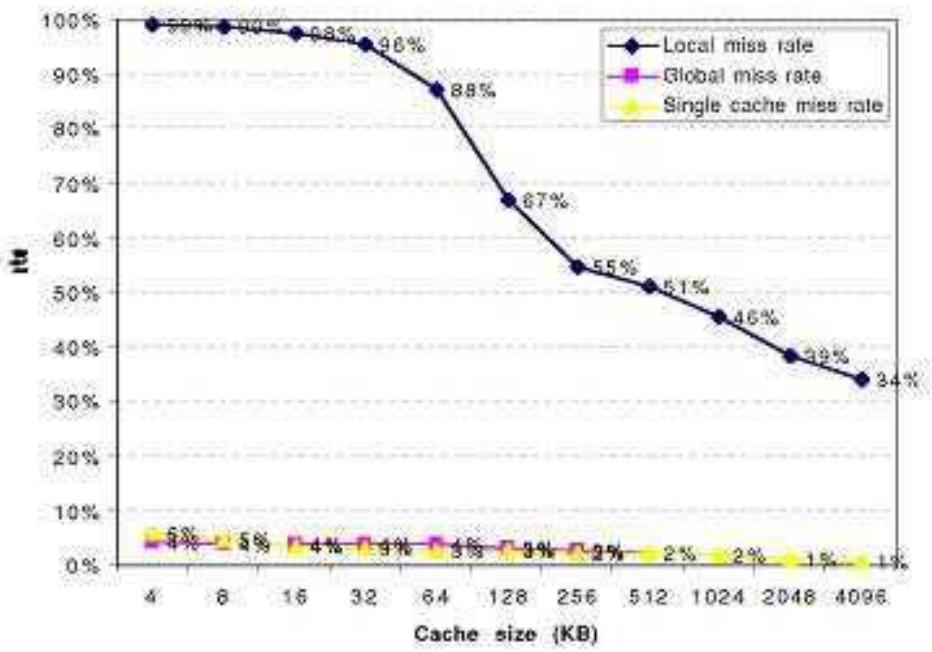
Cache Performance And various cache optimization categories.

- The average memory access time is calculated as follows
- Average memory access time = hit time + Miss rate x Miss Penalty.
- Where Hit Time is the time to deliver a block in the cache to the processor (includes time to determine whether the block is in the cache), Miss Rate is the fraction of memory references not found in cache (misses/references) and Miss Penalty is the additional time required because of a miss.
- The average memory access time due to cache misses predicts processor performance.
- First, there are other reasons for stalls, such as contention due to I/O devices using memory and due to cache misses
- Second, The CPU stalls during misses, and the memory stall time is strongly correlated to average memory access time. CPU time = (CPU execution clock cycles + Memory stall clock cycles) × Clock cycle time
- There are 17 cache optimizations into four categories:
- Reducing the miss penalty: multilevel caches, critical word first, read miss before write miss, merging write buffers, victim caches;
- Reducing the miss rate larger block size, larger cache size, higher associativity,pseudo-ssociativity, and compiler optimizations;
- Reducing the miss penalty or miss rate via parallelism: nonblocking caches, hardware prefetching, and compiler prefetching;
- 4 Reducing the time to hit in the cache: small and simple caches, avoiding address

Various techniques for Reducing Cache Miss Penalty

- The First Miss Penalty Reduction Technique follows the Adding another level of cache between the original cache and memory.

- The first-level cache can be small enough to match the clock cycle time of the fast CPU and the second-level cache can be large enough to capture many accesses that would go to main memory, thereby the effective miss penalty.
- The definition of average memory access time for a two-level cache. Using the subscripts L1 and L2 to refer, respectively, to a first-level and a second-level cache, the formula is
- Average memory access time = Hit timeL1 + Miss rateL1 × Miss penaltyL1 and Miss penaltyL1 = Hit timeL2 + Miss rateL2 × Miss penaltyL2 so Average memory access time = Hit timeL1 + Miss rateL1× (Hit timeL2 + Miss rateL2 ×Miss penaltyL2)
- **Local miss rate**—This rate is simply the number of misses in a cache divided by the total



Cache Performance

- Average memory access time
  - Timetotal mem access = Nhit×Thit + Nmiss×Tmiss
    - =Nmem access ×Thit + Nmiss ×Tmiss penalty
  - AMAT = Thit+ miss rate ×Tmiss penalty
- Miss penalty: time to replace a block from lower level, including time to replace in CPU
  - Access time: time to lower level(latency)
  - Transfer time: time to transfer block(bandwidth)
- Execution time: eventual optimization goal
  - CPU time = (busy cycles + memory stall cycles) ×Tcycle
- = IC ×(CPIexec+Nmiss per instr. ×Cyclemiss penalty) ×Tcycle
- = IC ×(CPIexec+miss rate.×(memory accesses / instruction) × Cyclemiss penalty) ×Tcycle

Performance Example

- Two data caches (assume one clock cycle for hit)
  - I: 8KB, 44% miss rate, 1ns hit time
  - II: 64KB, 37% miss rate, 2ns hit time
  - Miss penalty: 60ns, 30% memory accesses
  - CPIexec= 1.4
  - AMATI = 1ns + 44%×60ns = 27.4ns
  - AMATII = 2ns + 37%×60ns = 24.2ns
  - CPU timeI = IC×(CPIexec+30%×44%×(60/1))×1ns = 9.32IC
  - CPU timeII = IC×(CPIexec+30%×37%×(60/2))×2ns = 9.46IC
  - Larger cache ⇒smaller miss rate but longer Thit⇒reduced AMAT but not CPU time

Miss Penalty in OOO Environment

- In processors with out-of-order execution

  Memory accesses can overlap with other computation

  Latency of memory accesses is not always fully exposed

  E.g. 8KB cache, 44% miss rate, 1ns hit time, miss penalty: 60ns, only 70% exposed on average

  AMAT= 1ns + 44%×(60ns×70%) = 19.5ns

Cache Performance Optimizations

- Performance formulas
- AMAT = Thit+ miss rate ×Tmiss penalty
- CPU time = IC ×(CPIexec+miss rate.×(memory accesses / instruction) × Cyclemiss penalty) ×Tcycle
- Reducing miss rate
- Change cache configurations, compiler optimizations
- Reducing hit time
- Simple cache, fast access and address translation
- Reducing miss penalty
- Multilevel caches, read and write policies
- Taking advantage of parallelism
- Cache serving multiple requests simultaneously
- Perfecting

Classification of Cache Misses

Compulsory

- The first access to a block is never in the cache. Also called cold start misses or first reference misses (Misses in even an Infinite Cache)

Capacity

- If the cache cannot contain all the blocks needed during execution of a program, blocks must be discarded and later retrieved. (Misses in Fully Associative Size X Cache)

Conflict

- If block-placement strategy is set associative or direct mapped, blocks may be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses (Misses in N-way Associative, Size X Cache)

Cache Miss Rate

Three C's

Compulsory misses (cold misses)

The first access to a block: miss regardless of cache size

Capacity misses

Cache too small to hold all data needed

Conflict misses

More blocks mapped to a set than the associativity

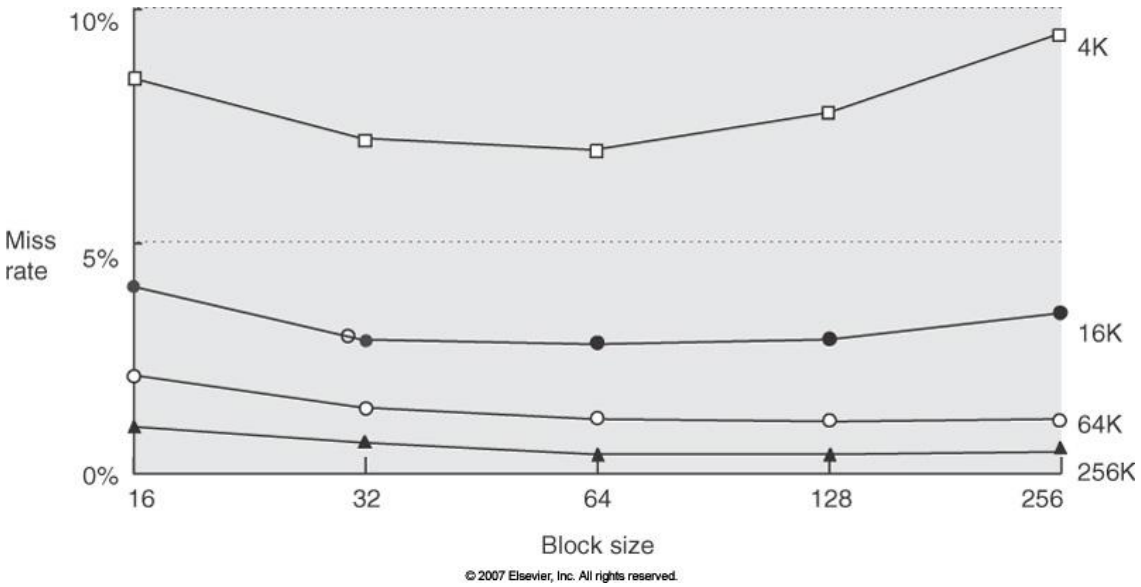
Reducing miss rate

Larger block size (compulsory)

Larger cache size (capacity, conflict)

Higher associativity (conflict)

Compiler optimizations (all three)



Reducing Cache Miss Rate

Larger blocks: compulsory misses reduced, but may increase conflict misses or even capacity misses if the cache is small; may also increase miss penalty

- Larger cache
- Less capacity misses
- Less conflict misses
- Implies higher associativity: less competition to the same set
- Has to balance hit time, energy consumption, and cost
- Higher associativity
- Less conflict misses

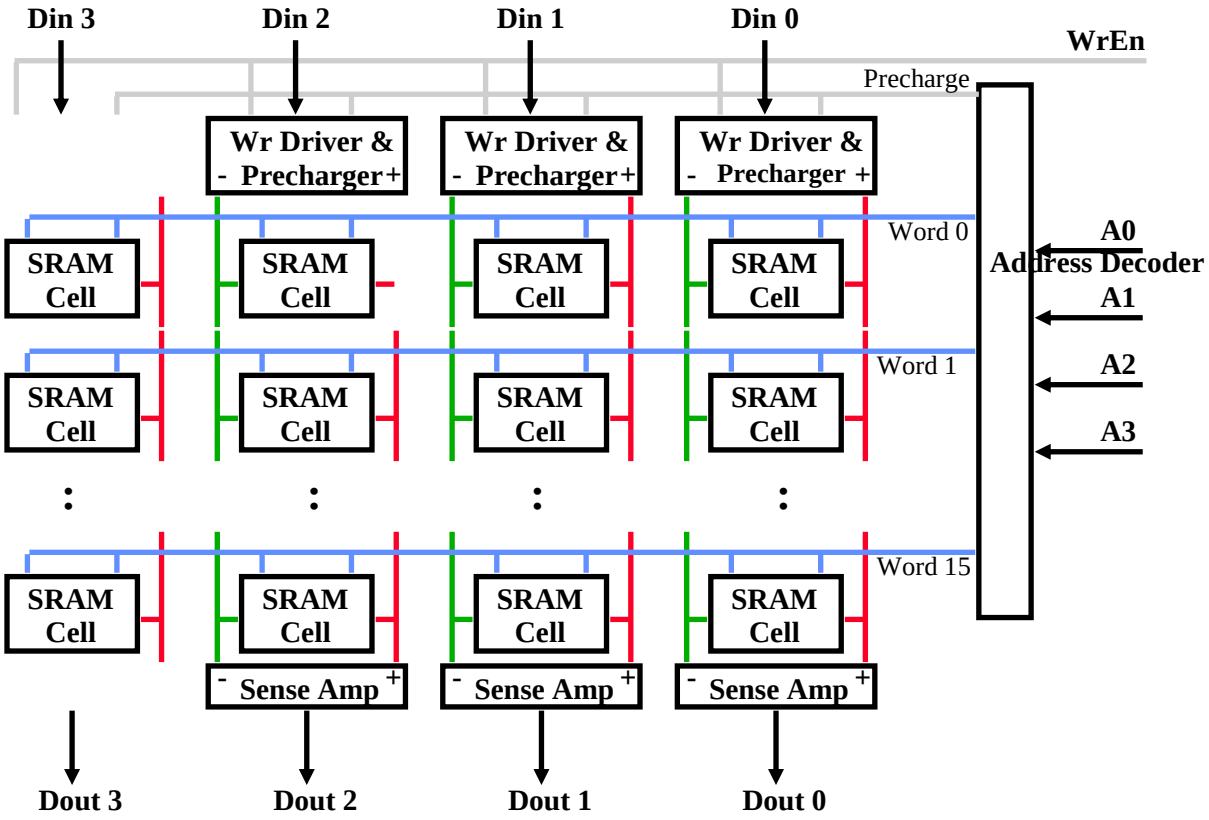
- ❖ Miss rate (2-way, X) ≈ Miss rate(direct-map, 2X)
- ❖ Similarly, need to balance hit time, energy consumption: diminishing return on reducing conflict misses

Main memory and performance

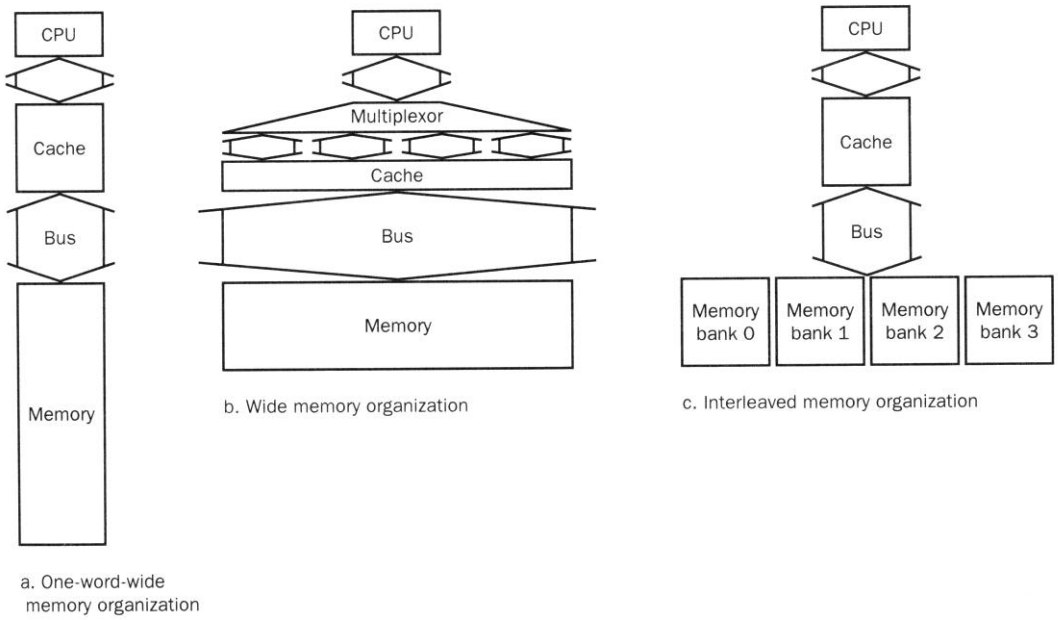
Performance of Main Memory:

- ➢ Latency: Cache Miss Penalty
    - **Access Time(AT***)*: time between request and word arrives
    - **Cycle Time(CT***)*: time between requests
  - o Bandwidth: I/O & Large Block Miss Penalty (L2)
- ➢ Main Memory, a 2D matrix, is DRAM:
  - o Dynamic since needs to be refreshed periodically (8 ms)
    - Difference in AT and CT, AT<CT
  - o Addresses divided into 2 halves, multiplexing them to memory:
    - **RAS or Row Access Strobe**
    - **CAS or Column Access Strobe**
- ➢ Cache uses SRAM:
  - o No refresh (6 transistors/bit vs. 1 transistor/bit)
    - No difference in AT and CT, AT=CT
  - o Address not divided

Typical SRAM Organization: 16-word x 4-bit



Main Memory Performance



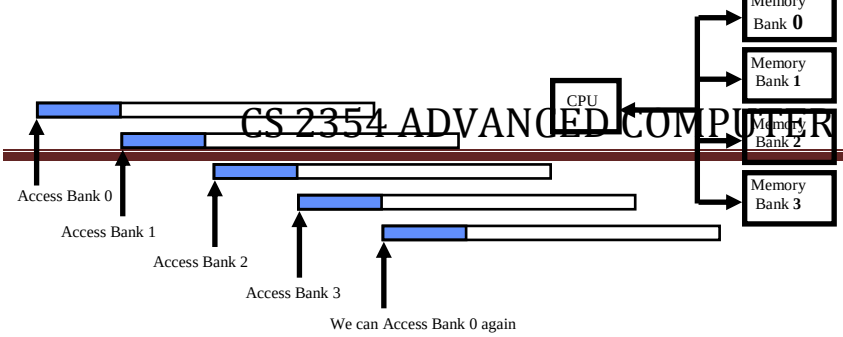
a. One-word-wide memory organization

b. Wide memory organization

c. Interleaved memory organization

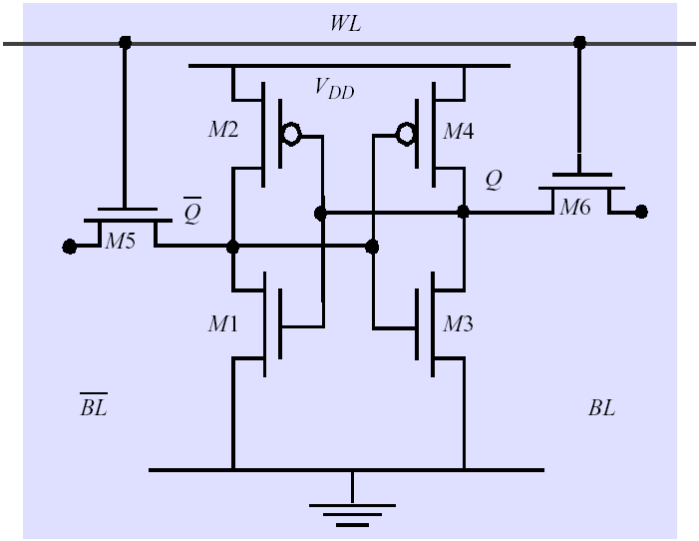
DRAM (Read/Write) Cycle Time >> DRAM (Read/Write) Access Time 2:1;

- **DRAM (Read/Write) Cycle Time:**
  - Analogy: A little kid can only ask his father for money on Saturday
- **DRAM (Read/Write) Access Time:**
  - Analogy: As soon as he asks, his father will give him the money
- DRAM Bandwidth Limitation analogy:
- **Timing model**
  - 1 cycle to send address,
  - 6 cycles to access data + 1 cycle to send data
  - Cache Block is 4 words
- Simple Mem. = 4 x (1+6+1) = 32
- Wide Mem. = 1 + 6 + 1 = 8
- Interleaved Mem. = 1 + 6 + 4x1 = 11



Static RAM (SRAM)

- Six transistors in cross connected fashion
- Provides regular AND inverted outputs
- Implemented in CMOS process



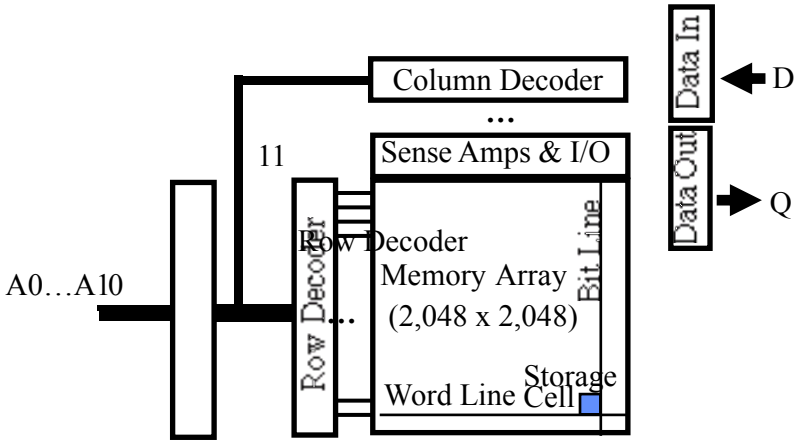
Dynamic RAM

- SRAM cells exhibit high speed/poor density
- DRAM: simple transistor/capacitor pairs in high density form

DRAM Operations

- **Write**
- Charge bitline HIGH or LOW and set wordline HIGH
- **Read**
- Bit line is precharged to a voltage halfway between HIGH and LOW, and then the word line is set HIGH.
- Depending on the charge in the cap, the precharged bitline is pulled slightly higher or lower.
- Sense Amp Detects change
- Need to sufficiently drive bitline
- Increase density => increase parasitic capacitance

DRAM logical organization (4 Mbit)



RAMBUS (RDRAM)

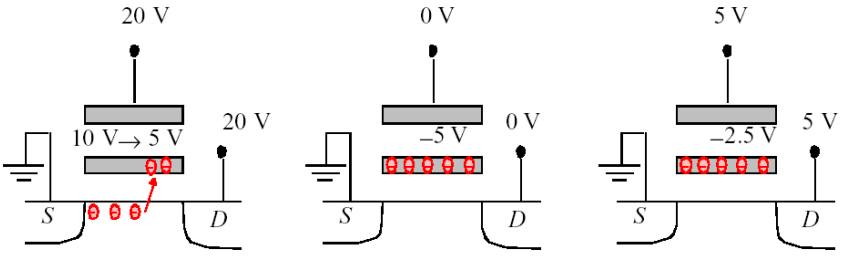
- Protocol based RAM w/ narrow (16-bit) bus
- High clock rate (400 Mhz), but long latency
- Pipelined operation
- Multiple arrays w/ data transferred on both edges of clock

Read-only memory (ROM)

- Programmed at time of manufacture
- Can not be written by the computer
- It is not erased by loss of power
- Some of them can be erased and rewritten by special hardware (EEPROM)
- One transistor / bit.
- Used in:
- BIOS of desktop computers
- Embedded devices (also serves as a code protection device)

FLASH Memory

- Floating gate transistor
- Presence of charge => "0"
- Erase Electrically or UV (EPROM)
- Performance
- Reads like DRAM (~ns)
- Writes like DISK (~ms). Write is a complex operation



Avalanche injection. Removing programming voltage leaves charge trapped. Programming results in higher V_T.

Types of storage devices

- **Primary storage:** is the storage provided by memory in a computer system e.g. ROM/RAM.
- **Secondary storage**: is storage provided by peripheral devices other than memory

Secondary storage is required in a computer system for three reasons

- The content of memory is usually volatile, which means that if power is disconnected the data is lost.
- The capacity in megabytes of memory is limited.
- **3.** Memory is more expensive than secondary storage.

Several types of disks may be used for Secondary storage.

- Floppy disks
- Hard disks
- Optical disks (including CD-ROM, writeable CD, DVD
- Backup Storage Devices e.g. tape

Floppy Disk

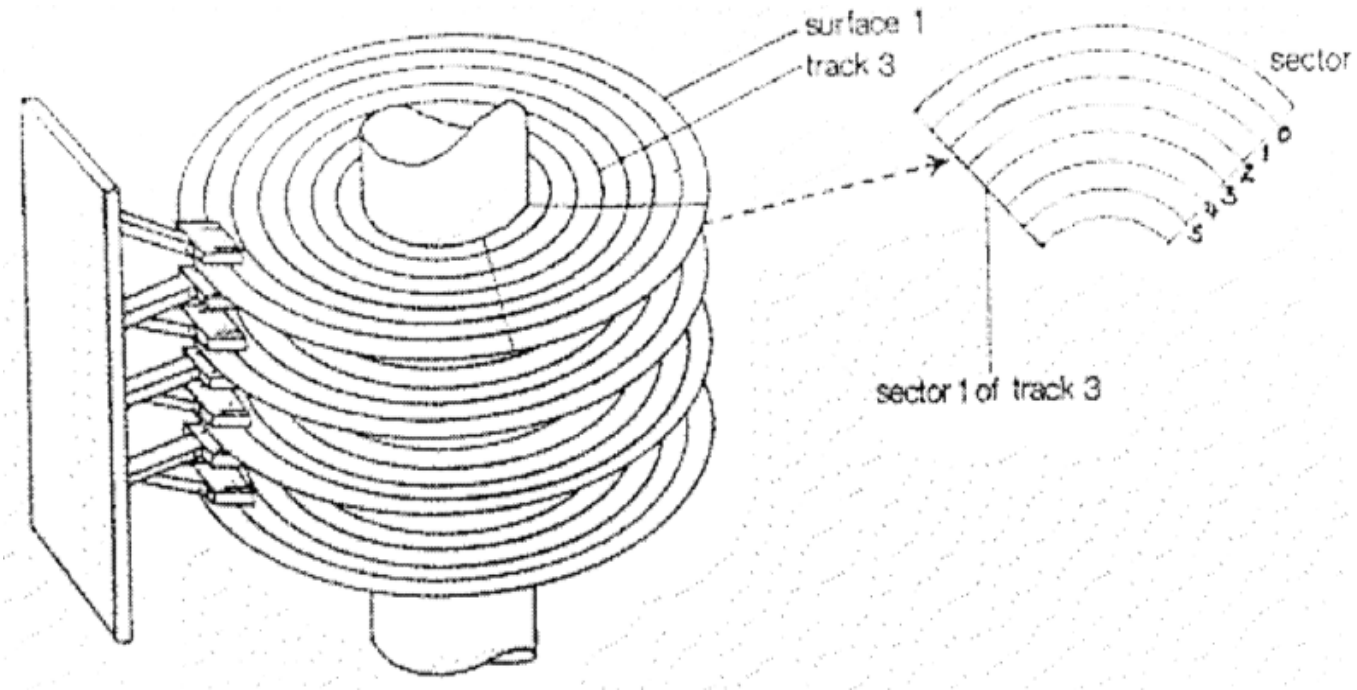
- A floppy disk is a low capacity disk which may be removed from the computer.
- There are two types:
- Those holding a small amount of data (typically 1.44 Mb)
- And 'Super floppies' known as ZIP disks (typically 100 Mb)
- Data may be written to and read from a floppy. A small notch can be used to make the disk read-only
- They are small lightweight and easy to transport.
- Ideal for backups of small amounts of data or for transfer of data from one machine to another.

- Floppy Drives are common to most if not all computers.
- On the down side, they may be easily misplaced, damaged or stolen.

There is a risk of transferring VIRUSES

Hard Disk

- A hard disk is a higher capacity medium, with up to hundreds of gigabytes.
- They are usually non-removable, but removable hard disks are becoming more common.
- They can be both read from and written to, and are the standard medium for storage on computer systems today.
- Hard disks are manufactured in metal and coated with a magnetisable recording medium, similar to the material used in a floppy disk or audio tape. Depending on the storage capacity of the unit, it may comprise a number of disks each having its own
- read/write head.
- Hard disks are much faster than floppy disks and can store much larger amounts of data.



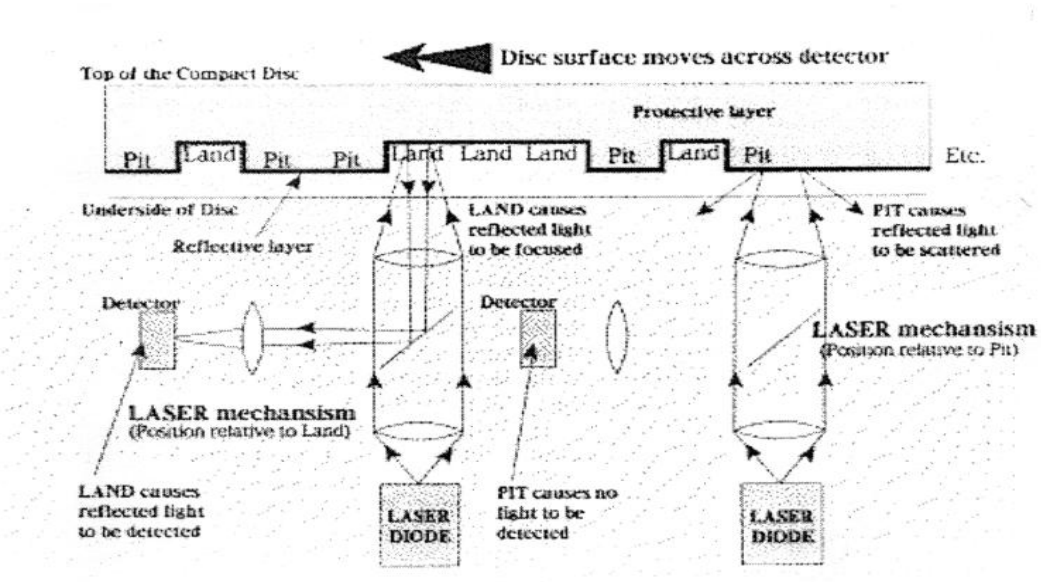
- You can see from the previous diagram a hard disk is made up of Sectors, Tracks and Cylinders.

The specification of a hard disk depends not only on its capacity but also:

Optical disks

- CD ROM = Compact Disk Read Only Memory, is an ideal device for storing large quantities of data and information such as large software packages.
- The CD drive uses laser technology to read the disk contents and therefore both access and transfer are extremely fast. With a typical capacity of 600 Mb they are used for software supply, reference material such as Encyclopaedias and games.

- DVD or Digital Versatile Disk is a higher capa city version of a CD and DVD drives have a higher transfer rate. DVD disks provide high quality playback of films and audio and are increasingly found as standard on the home PC. DVDs may be read only or read/write.
- They are sometimes known as DVD-ROM and DVD-RAM. DVD disks are double sided so data is stored on both sides of the disk. DVD technology uses a very shortwave laser beam to read pits from the spinning disk DVD disks typically holds 4.7 Gbytes of data Rewritable disks can be re-used thousands of times



Backup Storage Devices

- It is vital that all files stored in a computer system are backed up regularly.
- There are several high capacity devices.
- Cartridge tape back-up drives, which can hold up to
- 10 Gigabytes on a single tape.
- Zip disk drives, which hold 100 Megabytes.
- Jaz disk drives, which holds 1 or 2 Gigabytes
- Super floppy disk drives which can hold up to 120 Mb

CD writers, which hold 680 Megabytes

Buses

Buses-Definition

- A communication pathway connecting two or more devices
- Usually broadcast
- Often grouped
    - A number of channels in one bus
    - e.g. 32 bit data bus is 32 separate single bit channels
- Power lines may not be shown
- There are a number of possible interconnection systems
- Single and multiple BUS structures are most common
- e.g. Control/Address/Data bus (PC)
- e.g. Unibus (DEC-PDP)

Data Bus

- Carries data
    - Remember that there is no difference between "data" and "instruction" at this level Width is a key determinant of performance 8, 16, 32, 64 bit

Address bus

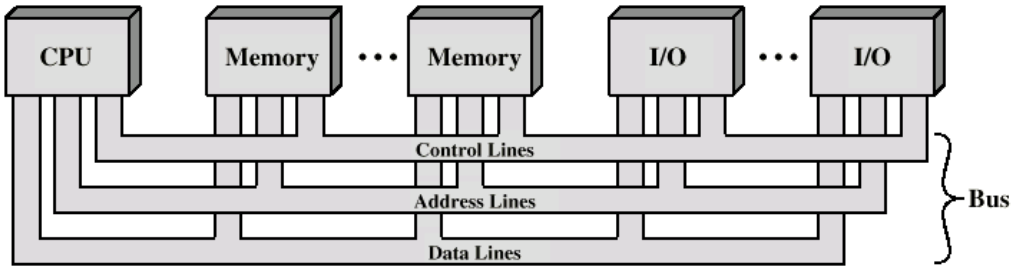
- Identify the source or destination of data **e.g**. CPU needs to read an instruction (data) from a given location in memory
- Bus width determines maximum memory capacity of system **e.g.** 8080 has 16 bit address bus giving 64k address space

Control Bus

Control and timing information

1. Memory read/write signal
2. Interrupt request
3. Clock signals

Bus Interconnection Scheme



Bus Types

- **Dedicated**

  Separate data & address lines
- **Multiplexed**
  - Shared lines
  - Address valid or data valid control line
  - Advantage - fewer lines
    - **Disadvantages**
    - More complex control
    - Ultimate performance
      - **Bus Arbitration**
      - More than one module controlling the bus
      - e.g. CPU and DMA controller
      - Only one module may control bus at one time
      - Arbitration may be centralised or distributed.

Centralised or Distributed Arbitration

Centralised

- Single hardware device controlling bus access
  - Bus Controller

- Arbiter
- May be part of CPU or separate

Distributed

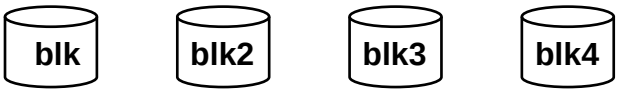
- Each module may claim the bus
- Control logic on all modules

RAIDs: Disk Arrays

Redundant Array of Inexpensive Disks

- Arrays of small and inexpensive disks
- Increase potential throughput by having many disk drives
- Data is spread over multiple disk
- Multiple accesses are made to several disks at a time
- Reliability is lower than a single disk
- But availability can be improved by adding redundant disks (RAID)
- Lost information can be reconstructed from redundant information
- MTTR: mean time to repair is in the order of hours
- MTTF: mean time to failure of disks is tens of years

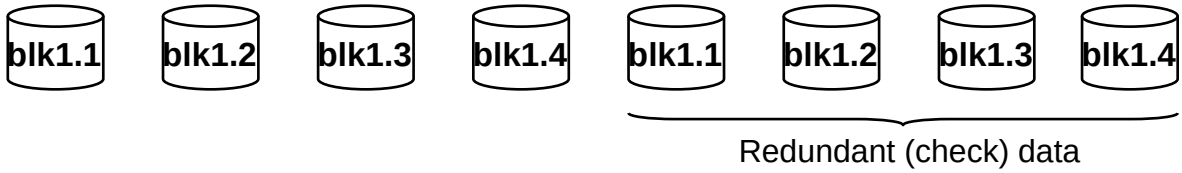
RAID: Level 0 (No Redundancy; Striping)



Multiple smaller disks as opposed to one big disk

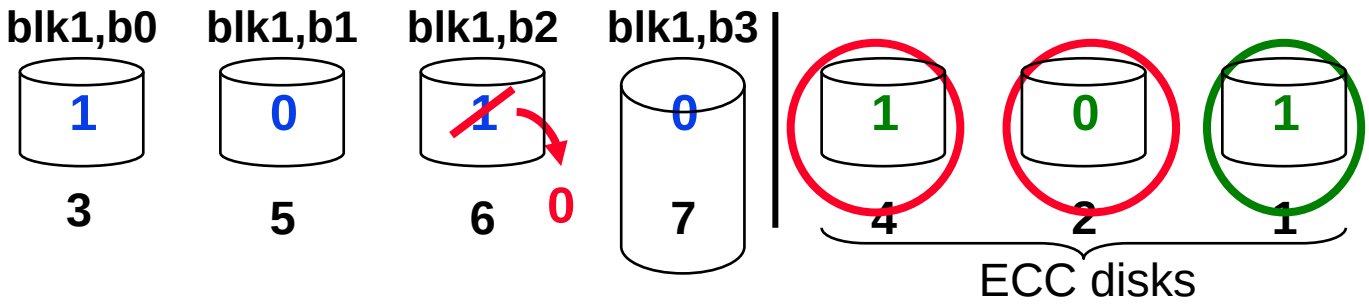
- Spreading the blocks over multiple disks – striping – means that multiple blocks can be accessed in parallel increasing the performance A 4 disk system gives four times the throughput of a 1 disk system
- Same cost as one big disk – assuming 4 small disks cost the same as one big disk
- Failure of one or more disks is more likely as the number of disks in the system increases.

RAID: Level 1 (Redundancy via Mirroring)



Uses twice as many disks as RAID 0 (e.g., 8 smaller disks with second set of 4 duplicating the first set) so there are always two copies of the data

- Redundant disks = # of data disks so twice the cost of one big disk writes have to be made to both sets of disks, so writes would be only 1/2 the performance of RAID 0. If a disk fails, the system just goes to the "mirror" for the data

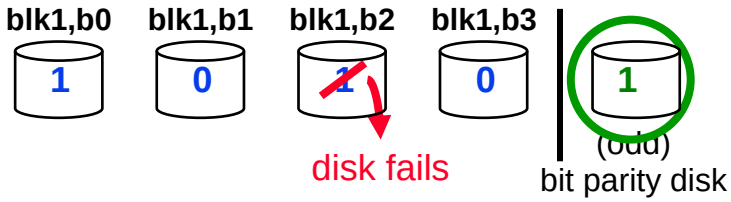


RAID: Level 2 (Redundancy via ECC)

ECC disks 4 and 2 point to either data disk 6 or 7 , but ECC disk 1 says disk 7 is okay, so disk 6 must be in error

- ECC disks contain the parity of data on a set of distinct overlapping disks Redundant disks = log (total # of data disks).
- so almost twice the cost of one big disk writes require computing parity to write to the ECC disks reads require reading ECC disk and confirming parity Can tolerate limited disk failure, since the data can be reconstructed

RAID: Level 3 (Bit-Interleaved Parity)



Cost of higher availability is reduced to 1/N where N is the number of disks in a protection group

- # Redundant disks = 1 × # of protection groups
- Writes require writing the new data to the data disk as well as computing the parity, meaning reading the other disks, so that the parity disk can be updated
- Can tolerate limited disk failure, since the data can be reconstructed

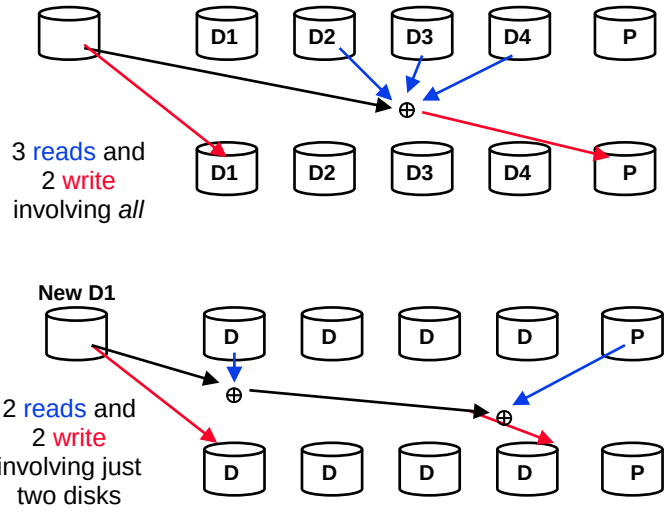
Reads require reading all the operational data disks as well as the parity disk to calculate the missing data that was stored on the failed disk

RAID: Level 4 (Block-Interleaved Parity)

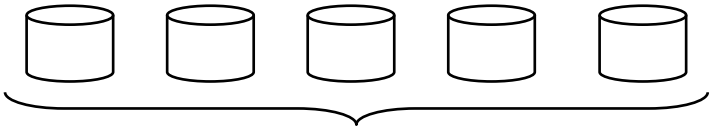
Cost of higher availability still only 1/N but the parity is stored as blocks associated with sets of data blocks

- Four times the throughput (striping)
- # redundant disks = 1 × # of protection groups
- Supports "small reads" and "small writes" (reads and writes that go to just one (or a few) data disk in a protection group)
  - by watching which bits change when writing new information, need only to change the corresponding bits on the parity disk
  - the parity disk must be updated on every write, so it is a bottleneck for back-to-back writes
  - Can tolerate *limited* disk failure, since the data can be reconstructed

RAID 3 small writes



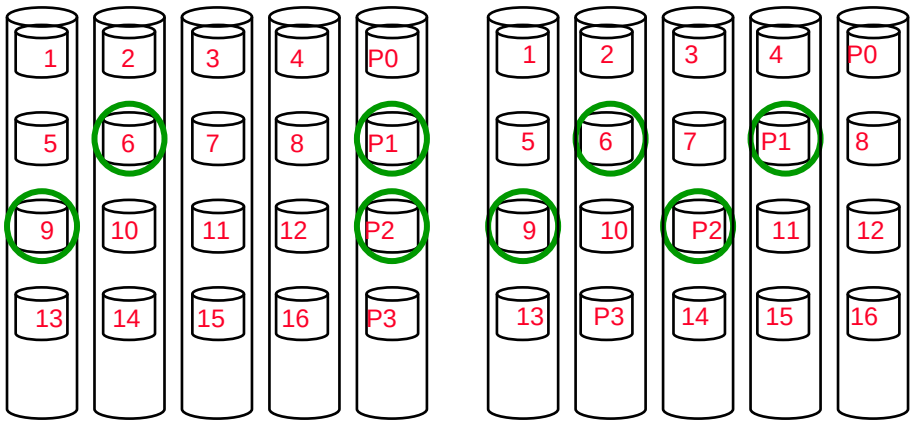
RAID: Level 5 (Distributed Block-Interleaved Parity)



- One of these assigned as the block parity disk

- Cost of higher availability still only 1/N but the parity block can be located on any of the disks so there is no single bottleneck for writes
- Still four times the throughput (striping)
- # redundant disks = 1 × # of protection groups
- Supports "small reads" and "small writes" (reads and writes that go to just one (or a few) data disk in a protection group)
- Allows multiple simultaneous writes as long as the accompanying parity blocks are not located on the same disk
- Can tolerate *limited* disk failure, since the data can be reconstructed

Distributing Parity Blocks



By distributing parity blocks to all disks, some small writes can be performed in parallel

Reliability, availability and dependability

- Reliability – measured by the mean time to failure (MTTF). Service interruption is measured by mean time to repair (MTTR)
- Availability – a measure of service accomplishment
- Availability = MTTF/(MTTF + MTTR)

- To increase MTTF, either improve the quality of the components or design the system to continue operating in the presence of faulty components
- Fault avoidance: preventing fault occurrence by construction
- Fault tolerance: using redundancy to correct or bypass faulty components (hardware)
    - Fault detection versus fault correction
    - Permanent faults versus transient faults

Input and Output Devices

- I/O devices are incredibly diverse with respect to
    - Behavior – input, output or storage
    - Partner – human or machine
    - Data rate – the peak rate at which data can be transferred between the I/O device and the main memory or processor

Device	Behavior	Partner	Data rate (Mb/s)
Keyboard	input	human	0.0001
Mouse	input	human	0.0038
Laser printer	output	human	3.2000
Graphics display	output	human	800.0000-8000.0000
Network/LAN	input or output	machine	100.0000-1000.0000
Magnetic disk	storage	machine	240.0000-2560.0000

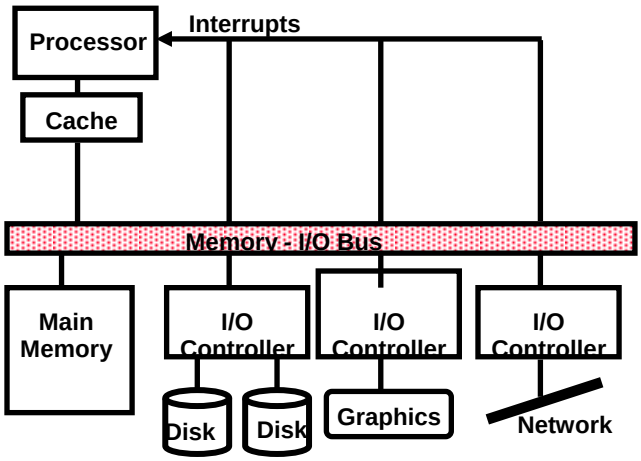
I/O Performance Measures

- I/O bandwidth (throughput) – amount of information that can be input (output) and communicated across an interconnect (e.g., a bus) to the processor/memory (I/O device) per unit time
- How much data can we move through the system in a certain time?
- How many I/O operations can we do per unit time?

- I/O response time (latency) – the total elapsed time to accomplish an input or output operation
- An especially important performance metric in real-time systems

- Many applications require both high throughput and short response times

A Typical I/O System



- Designing an I/O system to meet a set of bandwidth and/or latency constraints means

- Finding the weakest link in the I/O system – the component that constrains the design
  - The processor and memory system?
  - The underlying interconnection (e.g., bus) ?
  - The I/O controllers?
  - The I/O devices themselves?
- (Re)configuring the weakest link to meet the bandwidth and/or latency requirements
- Determining requirements for the rest of the components and (re)configuring them to support this latency and/or bandwidth

I/O System Performance Example

- A disk workload consisting of 64KB reads and writes where the user program executes 200,000 instructions per disk I/O operation and a processor that sustains 3 billion instr/s and averages 100,000 OS instructions to handle a disk I/O operation

- The maximum disk I/O rate (# I/O's/sec) of the processor is a memory-I/O bus that sustains a transfer rate of 1000 MB/s.

- Each disk I/O reads/writes 64 KB so the maximum I/O rate of the bus is SCSI disk I/O controllers with a DMA transfer rate of 320 MB/s that can accommodate up to 7 disks per controller disk drives with a read/write bandwidth of 75 MB/s and an average seek plus rotational latency of 6 ms.

- A disk workload consisting of 64KB reads and writes where the user program executes 200,000 instructions per disk I/O operation and a processor that sustains 3 billion instr/s and averages 100,000 OS instructions to handle a disk I/O operation
- The maximum disk I/O rate (# I/O's/s) of the processor is

$$\frac{\text{Instr execution rate}}{\text{Instr per I/O}} = \frac{3 \times 10^9}{(200 + 100) \times 10^3} = 10{,}000 \text{ I/O's/s}$$

- A memory-I/O bus that sustains a transfer rate of 1000 MB/s
- Each disk I/O reads/writes 64 KB so the maximum I/O rate of the bus is
- Bus bandwidth 1000 x 106
- Bytes per I/O 64 x 103
- SCSI disk I/O controllers with a DMA transfer rate of 320 MB/s that can accommodate up to 7 disks per controller
- Disk drives with a read/write bandwidth of 75 MB/s and an average seek plus rotational latency of 6 ms

Design principles

- Take advantage of parallelism
- Principle of locality
- Focus on the common case
- Amdahl's Law
- Generalized processor performance

1. Take advantage of parallelism

- Increasing throughput of server computer via multiple processors or multiple disks
- Detailed HW design
- Carry lookahead adders uses parallelism to speed up computing sums from linear to logarithmic in number of bits per operand
- Multiple memory banks searched in parallel in set-associative caches
- Pipelining: overlap instruction execution to reduce the total time to complete an instruction sequence.
- Not every instruction depends on immediate predecessor ⇒ executing instructions completely/partially in parallel possible

- **Classic 5-stage pipeline**:

  1) Instruction Fetch (I fetch),

  2) Register Read (Reg),

  3) Execute (ALU),

  4) Data Memory Access (Dmem),

  5) Register Write (Reg)

2.Principle of locality

- The Principle of Locality:
- Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
- Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
- Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon
  (e.g., straight-line code, array access)
- Last 30 years, HW relied on locality for memory perf.
- Guiding principle behind caches
- To some degree, guides instruction execution, too (90/10 rule)

3.Focus on the common case

- In making a design trade-off, favor the frequent case over the infrequent case
  - E.g., Instruction fetch and decode unit used more frequently than multiplier, so optimize it 1st
  - E.g., If database server has 50 disks / processor, storage dependability dominates system dependability, so optimize it 1st
- Frequent case is often simpler and can be done faster than the infrequent case

- E.g., overflow is rare when adding 2 numbers, so improve performance by optimizing more common case of no overflow
- May slow down overflow, but overall performance improved by optimizing for the normal case

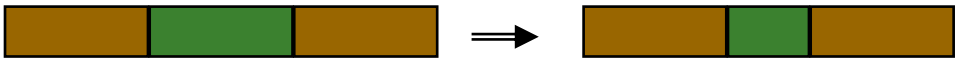
❖ What is frequent case and how much performance improved by making case faster => Amdahl's Law

4. Amdahl's Law

$$ExTime_{new} = ExTime_{old} \times \left[\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right]$$

$$Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$$Speedup_{maximum} = \frac{1}{\left(1 - Fraction_{enhanced}\right)}$$



5.Processor performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

UNIT V

UNIT V MULTI-CORE ARCHITECTURES
Software and hardware multithreading – SMT and CMP architectures – Design issues – Case studies – Intel Multi-core architecture – SUN CMP architecture - heterogeneous multi-core processors – case study: IBM Cell Processor.

Software and hardware multithreading

- The ability of an operating system to execute different parts of a program, called threads, simultaneously.

- The programmer must carefully design the program in such a way that all the threads can run at the same time without interfering with each other

Two levels of thread

- User level (for user thread)
- Kernel level(for kernel thread)

User threads

- User threads are supported above the kernel and are implemented by a thread library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel.
- Because the kernel is unaware of user-level threads, all thread creation and scheduling are done in user space without the need for kernel intervention.
- User-level threads are generally fast to create and manage User-thread libraries include POSIX Pthreads, Mach C-threads,and Solaris 2 UI-threads.

Kernel level

- Kernel threads are supported directly by the operating system: The kernel performs thread creation, scheduling, and management in kernel space.
- Because thread management is done by the operating system, kernel threads are generally slower to create and manage than are user threads.

- ❖ Most operating systems-including Windows NT, Windows 2000, Solaris 2, BeOS, and Tru64 UNIX (formerly Digital UN1X)-support kernel threads.

Multi threading Models

There are three models for thread libraries, each with its own trade-offs

- Many threads on one LWP (many-to-one)
- One thread per LWP (one-to-one)
- Many threads on many LWPs (many-to-many)

Many-to-one

The many-to-one model maps many user-level threads to one kernel thread. Advantages: Totally portable More efficient Disadvantages: cannot take advantage of parallelism The entire process is block if a thread makes a blocking system call Mainly used in language systems, portable libraries like solaris 2

One-to-one

The one-to-one model maps each user thread to a kernel thread. Advantages: allows parallelism Provide more concurrency Disadvantages: Each user thread requires corresponding kernel thread limiting the number of total threads Used in Linux Threads and other systems like Windows 2000, Windows NT

Many-to-many

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. Advantages: Can create as many user thread as necessary Allows parallelism Disadvantages: kernel thread can the burden the performance Used in the Solaris implementation of Pthreads

SMT and CMP architectures

- ❖ Simultaneous multithreading (SMT) is one of the two main implementations of multithreading, the other form being temporal multithreading.

- In temporal multithreading, only one thread of instructions can execute in any given pipeline stage at a time. In simultaneous multithreading, instructions from more than one thread can be executing in any given pipeline stage at a time.
- This is done without great changes to the basic processor architecture: the main additions needed are the ability to fetch instructions from multiple threads in a cycle, and a larger register file to hold data from multiple threads.
- The number of concurrent threads can be decided by the chip designers, but practical restrictions on chip complexity have limited the number to two for most SMT implementations.
- Because the technique is really an efficiency solution and there is inevitable increased conflict on shared resources, measuring or agreeing on the effectiveness of the solution can be difficult.
- However, measured energy efficiency of SMT with parallel native and managed workloads on historical 130 nm to 32 nm Intel SMT (Hyper-Threading) implementations found that in 45 nm and 32 nm implementations, SMT is extremely energy efficient, even with inorder Atom processors [ASPLOS'11]. In modern systems, SMT effectively exploits concurrency with very little additional dynamic power.
- That is, even when performance gains are minimal the power consumption savings can be considerable.
- Some researchers have shown that the extra threads can be used to proactively seed a shared resource like a cache, to improve the performance of another single thread, and claim this shows that SMT is not just an efficiency solution. Others use SMT to provide redundant computation, for some level of error detection and recovery.
- However, in most current cases, SMT is about hiding memory latency, increasing efficiency, and increasing throughput of computations per amount of hardware used.

CMP Architectures

- Superscalar means executing multiple instructions at the same time while chip-level multithreading (CMT) executes instructions from multiple threads within one processor chip at the same time. There are many ways to support more than one thread within a chip, namely:
- Interleaved multithreading: Interleaved issue of multiple instructions from different threads, also referred to as temporal multithreading.
- It can be further divided into fine-grain multithreading or coarse-grain multithreading depending on the frequency of interleaved issues. **Fine-grain** multithreading—such as in a barrel processor—issues instructions for different threads after every cycle, while **coarse-grain** multithreading only

switches to issue instructions from another thread when the current executing thread causes some long latency events (like page fault etc.).

- Coarse-grain multithreading is more common for less context switch between threads. For example, Intel's Montecito processor uses coarse-grain multithreading, while Sun's UltraSPARC T1 uses fine-grain multithreading. For those processors that have only one pipeline per core, interleaved multithreading is the only possible way, because it can issue at most one instruction per cycle.
- Simultaneous multithreading (SMT): Issue multiple instructions from multiple threads in one cycle. The processor must be superscalar to do so.
- Chip-level multiprocessing (CMP or multicore): integrates two or more processors into one chip, each executing threads independently.
- Any combination of multithreaded/SMT/CMP.
- The key factor to distinguish them is to look at how many instructions the processor can issue in one cycle and how many threads from which the instructions come.
- For example, Sun Microsystems' UltraSPARC T1 (known as "Niagara" until its November 14, 2005 release) is a multicore processor combined with fine-grain multithreading technique instead of simultaneous multithreading because each core can only issue one instruction at a time

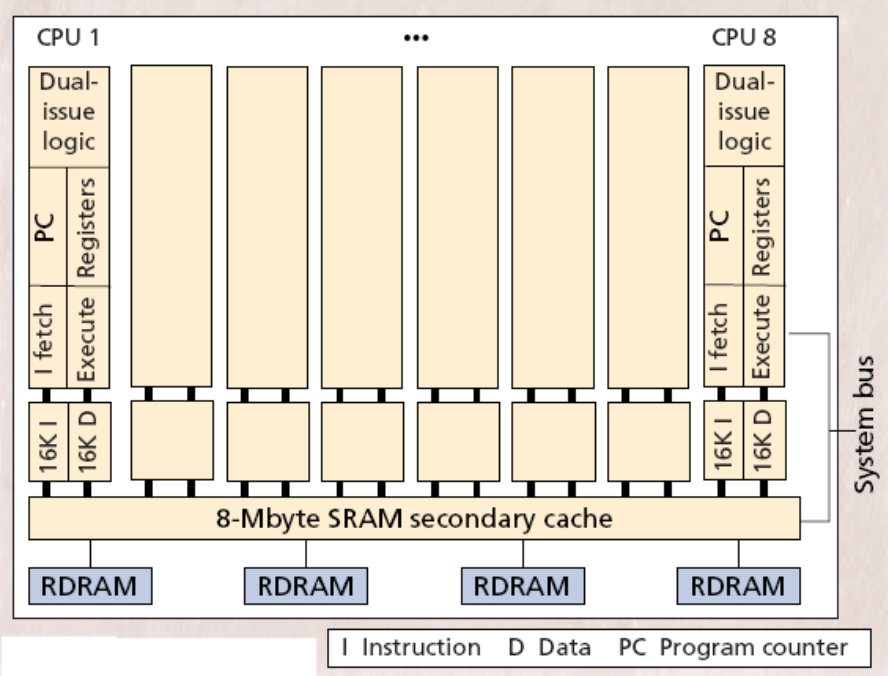
- Single-core microprocessor performance increases are beginning to slow [1] due to:
  - Increasing power consumption (>100 W)
  - Increasing heat dissipation
  - Diminishing performance gains from ILP & TLP
- As a result manufactures are turning to a multi-core microprocessor approach
  - Multiple smaller energy efficient processing cores are integrated onto a single chip
  - Improves overall performance by performing more work concurrently
  - The latencies associated with chip-to-chip communication disappear, Shared data structures are much less of a problem.

CMP Architectures

- Two general types of multi-core or chip multiprocessor (CMP) architectures
  - Homogeneous CMPs – all processing elements (PEs) are the same
  - Heterogeneous CMPs – comprised of different PEs

- Homogenous dual-core processors for PCs are now available from all major manufactures
- Heterogeneous CMPs are available in the form of multiprocessor systems-on-chips (MPSoCs)

Single chip Multiprocessor architecture



CMP Advantages

- CMPs have several advantages over single processor solutions
  - Energy and silicon area efficiency
    - By Incorporating smaller less complex cores onto a single chip
    - Dynamically switching between cores and powering down unused cores [5]
  - Increased throughput performance by exploiting parallelism
    - Multiple computing resources can take better advantage of instruction, thread, and process level parallelism

Design Issues

- **Superscalar technique**: which tries to increase Instruction level parallelism (ILP) by executing multiple instructions at the same time (termed: simultaneously); by "simultaneously" dispatching instructions (termed:

instruction dispatching) to multiple redundant execution units built inside the processor.

- **Chip-level multithreading (**CMT**) technique**: using Thread level parallelism (TLP) in order to executes instructions from multiple threads within one processor chip at the same time.
- There are many ways to support more than one thread inside a chip, namely:
- **Interleaved multithreading (IMT)** : Interleaved issue of multiple instructions from different threads, also referred to as Temporal multithreading. It can be further divided into fine-grain multithreading or coarse-grain multithreading depending on the frequency of interleaved issues. **Fine-grain** multithreading issues instructions for different threads after every cycle, while **coarse-grain** multithreading only switches to issue instructions from another thread when the current executing thread causes some long latency events (like page fault etc.). Coarse-grain multithreading is more common for less context switch between threads. For processors with one pipeline per core, interleaved multithreading is the only possible way, because it can only issue up to one instruction per cycle.
- **Simultaneous multithreading (SMT)**: Issue multiple instructions from multiple threads in one cycle. The processor must be superscalar to do so.
- **Chip-level multiprocessing (CMP or Multi-core processor)**: integrates two or more superscalar processors into one chip, each executes threads independently.

Intel Multi-core architecture

- The **Intel Core microarchitecture** (previously known as the **Next-Generation Micro-Architecture**) is a multi-core processor microarchitecture unveiled by Intel in Q1 2006.]
- It is based on the Yonah processor design and can be considered an iteration of the P6 microarchitecture, introduced in 1995 with Pentium Pro.
- The high power consumption and heat intensity, the resulting inability to effectively increase clock speed, and other shortcomings such as the inefficient pipeline were the primary reasons for which Intel abandoned the NetBurst microarchitecture and switched to completely different architectural design, delivering high efficiency through a small pipeline rather than high clock speeds. It is

worth noting that the Core microarchitecture never reached the clock speeds of the Netburst microarchitecture, even after moving to the 45 nm lithography.

- The first processors that used this architecture were code-named **Merom**, **Conroe**, and **Woodcrest**; Merom is for mobile computing, Conroe is for desktop systems, and Woodcrest is for servers and workstations.
- While architecturally identical, the three processor lines differ in the socket used, bus speed, and power consumption. Mainstream Core-based processors are branded *Pentium Dual-Core* or *Pentium* and low end branded *Celeron*; server and workstation Core-based processors are branded *Xeon*, while desktop and mobile Core-based processors are branded as *Core 2*.
- Despite their names, processors sold as Core Solo/Core Duo and Core i3/i5/i7 do not actually use the Core microarchitecture and are based on the Enhanced Pentium M and newer Nehalem/Sandy Bridge/Haswell microarchitectures, respectively.

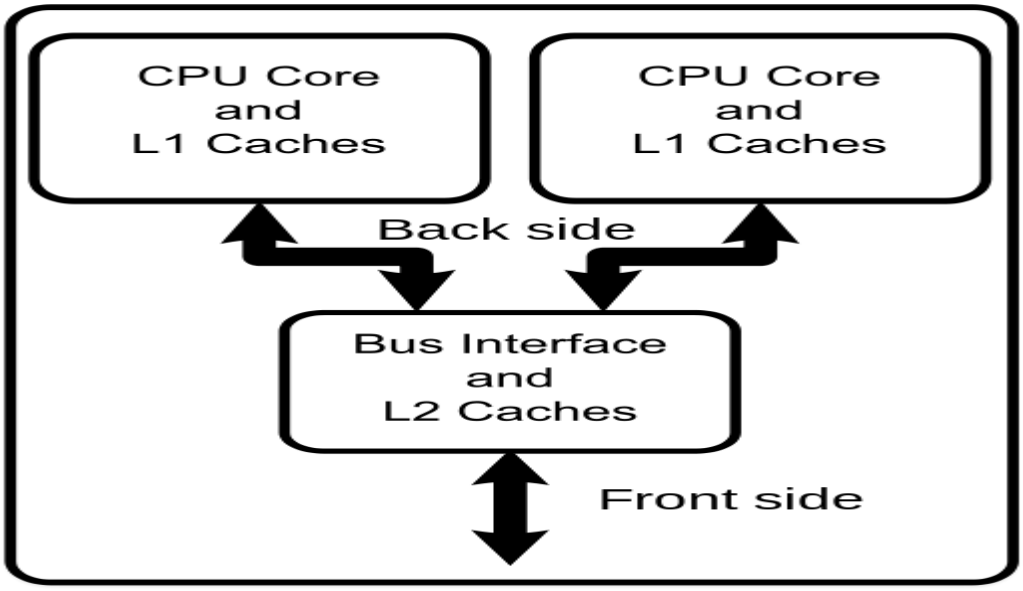
Heterogeneous multi-core processors

- A **multi-core processor** is a single computing component with two or more independent actual central processing units (called "cores"), which are the units that read and execute program instructions.
- The instructions are ordinary CPU instructions such as add, move data, and branch, but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing.
- Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package.
- Processors were originally developed with only one core. A dual-core processor has two cores (e.g. AMD Phenom II X2, Intel Core Duo), a quad-core processor contains four cores (e.g. AMD Phenom II X4, Intel's quad-core processors, see i5, and i7 at Intel Core), a 6-core processor contains six cores (e.g. AMD Phenom II X6, Intel Core i7 Extreme Edition 980X), an 8-core processor contains eight cores (e.g. Intel Xeon E7-2820, AMD FX-8350), a 10-core processor contains ten cores (e.g. Intel Xeon E7-2850), a 12-core processor contains twelve cores.
- A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely.
- For example, cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods.

- Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar.

Homogeneous multi-core systems

- Homogeneous multi-core systems include only identical cores, heterogeneous multi-core systems have cores that are not identical.
- Just as with single-processor systems, cores in multi-core systems may implement architectures such as superscalar, VLIW, vector processing, SIMD, or multithreading.
- Multi-core processors are widely used across many application domains including general-purpose, embedded, network, digital signal processing (DSP), and graphics.
- The improvement in performance gained by the use of a multi-core processor depends very much on the software algorithms used and their implementation.
- In particular, possible gains are limited by the fraction of the software that can be run in parallel simultaneously on multiple cores; this effect is described by Amdahl's law.
- In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache(s), avoiding use of much slower main system memory.
- Most applications, however, are not accelerated so much unless programmers invest a prohibitive amount of effort in re-factoring the whole problem.[3] The parallelization of software is a significant ongoing topic of research.



IBM Cell Processor

- The **IBM** microprocessor is a chip made by IBM for their zEnterprise 196 mainframe computers, announced on July 22, 2010.
- The processor was developed over a three year time span by IBM engineers from Poughkeepsie, New York; Austin, Texas; and Böblingen, Germany at a cost of US$1.5 billion. Manufactured at IBM's Fishkill, New York fabrication plant, the processor began shipping on September 10, 2010. IBM stated that it was the world's fastest microprocessor at the time.
- The chip measures 512.3 mm^2 and consists of 1.4 billion transistors fabricated in IBM's 45 nm CMOS silicon on insulator fabrication process, supporting speeds of 5.2 GHz: at the time, the highest clock speed CPU ever produced for commercial sale.
- The processor implements the CISC z/Architecture with a new superscalar, out-of-order pipeline and 100 new instructions. The instruction pipeline has 15 to 17 stages; the instruction queue can hold 40 instructions; and up to 72 instructions can be "in flight". It has four cores, each with a private 64 KB L1 instruction cache, a private 128 KB L1 data cache and a private 1.5 MB L2 cache.
- In addition, there is a 24 MB shared L3 cache implemented in eDRAM and controlled by two on-chip L3 cache controllers. There's also an additional shared L1 cache used for compression and cryptography operations.
- Each core has six RISC-like execution units, including two integer units, two load-store units, one binary floating point unit and one decimal floating point unit.
- The z196 chip can decode three instructions and execute five operations in a single clock cycle.
- The z196 chip has on board DDR3 RAM memory controller supporting a RAID like configuration to recover from memory faults.
- The z196 also includes a GX bus controller for accessing host channel adapters and peripherals. Additionally, each chip includes co-processors for cryptographic and compression functionality.

Shared Cache

- Even though the z196 processor has on-die facilities for symmetric multiprocessing (SMP), there are 2 dedicated companion chips called the Shared Cache (SC) that each adds 96 MB off-die L4 cache for a total of 192 MB L4 cache.
- L4 cache is shared by all processors in the book. The SC chip consists of 1.5 billion transistors and measures 478.8 mm^2, manufactured with the same 45 nm process as the z196 chip.
- Each chip also has 24 MB L3 cache shared by the 4 cores on the chip.

Multi-chip module

- The zEnterprise System z196 uses multi-chip modules (MCMs) which allows for six z196 chips to be on a single module.
- Each MCM has two shared cache chips allowing processors on the MCM to be connected with 40 GB/s links.
- The different models of the zEnterprise System have a different number of active cores. To accomplish this, some processors in each MCM may have its fourth core