ELC303A—Computer 3 A
Electronics and Electrical Communications Department
Undergraduate Studies

Aim

By the end of the class the student should be able to:

1. recognize, explain, and analyze the microarchitecture features of current processor designs,
2. indicate the advantages and disadvantages of the various interconnection schemes used to connect these processors and other system components especially memories and input/output devices,

   and
3. design simple interface circuits (including analog to digital and digital to analog conversion) between processors and other system components.

Introduction

This class builds on the previous classes dealing with digital logic design and micorprocessors. The microprocessors studied thus far are basic simple units. The structures studied exist in current microcontrollers and simple processors used in embedded systems.

Advanced processors use more elaborate ideas. In this class, we study the remaining important techniques: pipelines, branch prediction, and caches. These techniques are widely used in all high performance digital systems and not just processors.

Once we finish the internals of the processor, we start exploring what lies beyond the processor: buses, serial/parallel interconnects, input/output interfacing, ...

General outline *(tentative)*

Pipelines: Pipelines, exception handling, branch prediction, multiple issue machines, static and dynamic scheduling. (5 lectures)

Memories: memory types, caches, interleaving, virtual memory, translation look-aside buffers. (5 lectures)

Interconnections: serial versus parallel, serial and parallel bus standards, arbitration, split bus transactions, multiple masters, UART, USRT, USART, PCI, SATA, SCSI. (4 lectures)

D/A and A/D: Analog interfacing, interfacing D/As, interfacing A/Ds.(4 lectures)

Assessment *(tentative)*

Quiz and classwork	10%
Midterm	20%
Final exam	70%

Lecture 1: Pipelines

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

Performance

Performance is usually measured using one of two ways:

latency: the time delay taken to finish the task or

throughput: the number of tasks finished in a fixed time.

Are they the inverse of each other?
If not are they completely independent?
Real life examples:

- baking bread,
- passengers in a train station, or
- moving to and from the university using a bus versus a car.

Depending on the application for which we optimize the design we choose the appropriate measure. (Usually throughput.)

Moving one step at a time

A fire started in a farm. There is a nearby well and some buckets.

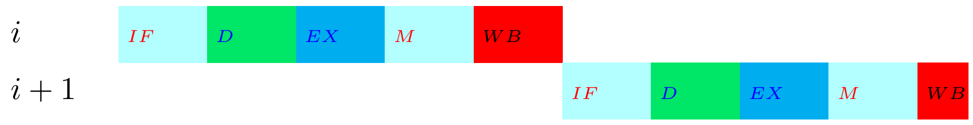
- Only one person fills a bucket, runs to the fire, throws the water, and goes back to repeat. *One entity does everything.*
- A group is there and forms a line. One fills a bucket and those in the line move the filled bucket forward to the one near the fire to extinguish it. When the bucket is empty, the line returns it to be refilled. *Each does a specific job but the flow is not continuous.*
- Even better: while the bucket is moving forward, the one near the water fills another bucket and the process repeats. *Each does a specific job. The flow is continuous due to parallelism in the action.*

(We plan to extinguish the fire with multiple lines in future lectures!)

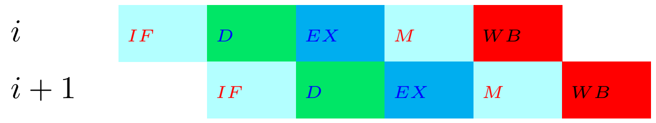
Processor pipelines

One long cycle: $\boxed{\text{IF, D, EX, Mem, WB}}$

Several short cycles: One instruction at a time.



Pipelined: With an overlap there is a pipeline.



The time taken by each instruction is *not* decreased but we have a much higher throughput and the program ends in a shorter time. (Remember the fire.)

Ideal speedup

There is nothing magical about five stages. Some processors have more than twenty! For a processor with n stages,

- without a pipeline, we start a new instruction each n clock cycles;
- with a pipeline, we start a new instruction every clock cycle.

For N instructions in the program and a clock cycle time of τ we have a speedup of

$$\frac{N \times n \times \tau}{N \times 1 \times \tau} = n.$$

$\Rightarrow$ divide the task into a large number of stages (superpipeline) to get a higher clock frequency and hence a higher throughput. *This is not completely true.*

Down to reality

- It is easier to decode while reading the registers if the instruction format has fixed field sizes. With a varying width format pipelining is harder.
- The pipeline latches add some time overhead.
- What if we need to access memory for data and for instructions simultaneously? *This is a structural hazard.*
- What if a hardware device sends an interrupt to the processor? *We must handle exceptions correctly.*
- Somethings are not equal: a floating point division takes longer than an integer addition! *Some instructions take more cycles.*

Dependences and hazards

Dependence: when instructions in the program are dependent

- they use the same hardware, (structural)
- they use the same data storage, (data)
- one instruction indicates whether the other one is executed or not. (control)

Hazard: two dependent instructions are in the pipeline together.

For structural and data hazards we may *stall* the following instructions. For control hazards we *flush* the pipeline.

Structural hazards

For a simple system we may get

Ld	*IF*	*D*	*EX*	*M*	*WB*				
$i + 1$		*IF*	*D*	*EX*	*M*	*WB*			
$i + 2$			*IF*	*D*	*EX*	*M*	*WB*		
$i + 3$				*IF*	*D*	*EX*	*M*	*WB*	
$i + 4$					*IF*	*D*	*EX*	*M*	*WB*

which we transform to (*Do you see another solution?*)

Ld	*IF*	*D*	*EX*	*M*	*WB*					
$i + 1$		*IF*	*D*	*EX*	*M*	*WB*				
$i + 2$			*IF*	*D*	*EX*	*M*	*WB*			
$i + 3$				stall	*IF*	*D*	*EX*	*M*	*WB*	
$i + 4$						*IF*	*D*	*EX*	*M*	*WB*

Avoiding structural hazards

- The replication of the resource
  - gives a good performance
  - but increases the area and may have some timing overheads.

  Good for cheap resources or highly contended ones (such as caches).
- The use of a pipeline in the contended resource
  - gives a good performance with a small area
  - but is sometimes complicated to implement (example RAM).

  Good for divisible multi-cycle resources.

Design to minimize structural hazards

A better design might help

1. Each instruction uses a given resource *only once*.
2. Each instruction uses a given resource *for one cycle*.
3. All instructions use a given resource *in the same stage*.

This is why ALU instructions go through the *M* stage although they do not do anything in it.

- This may be less than optimal!
- Some instructions (ex: divide) are much longer than others.

Points to take home

- Revise your previous knowledge
- Look around you for exciting examples of performance improvement
- Think about the problems of pipelines

Lecture 2: Exceptions everywhere

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

Where are we?

- It is easier to pipeline instructions that have a fixed format:
  - all the instructions are of the same size
    and
  - in all instructions the fields have a fixed size and occupy fixed locations.
- The use of more pipeline stages increases the frequency of operation but it adds
  - timing overheads
    and
  - more hazards.

Data hazards

What are the dependencies that you see in the following code?

I_1: `DIV R3, R1, R2`
I_2: `ADD R5, R3, R2`
I_3: `MUL R1, R2, R6`
I_4: `ADD R5, R1, R5`
I_5: `MUL R4, R2, R6`

Types of dependencies

If instruction i precedes instruction j and the sources or destinations match then we have a dependency.

	D_i	S_{1_i} or S_{2_i}
S_{1_j} or S_{2_j}	Essential, RAW	RAR
D_j	Output, WAW	Ordering, WAR

Bypass instead of RAW stalls

It is better to *forward* (or *bypass*) the data instead of stalling.

Add R5, R3, R2	IF	D	EX	M	WB			
Sub R6, R5, R1		IF	D	EX	M	WB		
Add R4, R5, R7			IF	D	EX	M	WB	
Add R8, R5, R4				IF	D	EX	M	WB

Some must stall

Unfortunately, we cannot always bypass

Ld R5, 0(R3)	IF	D	EX	M	WB				
Sub R6, R5, R1		IF	D	stall	EX	M	WB		
Add R4, R5, R7			IF	stall	D	EX	M	WB	
Add R8, R5, R4				stall	IF	D	EX	M	WB

Control hazards

- For these we must flush any instructions from the wrong direction
- We will deal with “prediction” in the coming few lectures.

Extending the basic pipeline

We started by forcing all the integer instructions to pass through the same number of stages even if they do not use them. *Why?*
However, the execution of a double precision floating point divide takes from 4 (most aggressive techniques) to over 50 (simple algorithms) cycles.

- Extend the clock cycle. Everything is slow!
- Allow some instructions to take multiple cycles in their execution.

Multi-cycle instructions

Assume multiply takes 5 cycles and add takes 3 cycles.

Ld.D F4, 0(R2)	*IF*	*D*	*EX*	*M*	*WB*									
Mul.D F0, F4, F6		*IF*	*D*	stall	EX_{M1}	EX_{M2}	EX_{M3}	EX_{M4}	EX_{M5}	*M*	*WB*			
Ld.D F8, 0(R3)			*IF*	stall	*D*	*EX*	*M*	*WB*						
Add.D F6, F4, F6					*IF*	*D*	EX_{A1}	EX_{A2}	EX_{A3}	*M*	*WB*			
Add.D F2, F0, F8						*IF*	*D*	$stall_1$	$stall_2$	EX_{A1}	EX_{A2}	EX_{A3}	*M*	*WB*
S.D F2, 0(R2)							*IF*	$stall_1$	$stall_2$	*D*	*EX*	$stall_3$	*M*	*WB*

- A specific unit deals with each of the extended instructions.
- Multi-cycle instructions increase the number of stall cycles.
- Now, we get in order start but out of order termination.
- We may also get multiple instructions in the *M* or *WB* stage. $\Rightarrow$ Stall either at the *D* or at the *WB* stage.
  Is it really necessary to stall?

The exceptions

We have external interrupts and internal exceptions. These events have several classifications.

1. User requested versus coerced.
2. Maskable versus nonmaskable.
3. Terminate versus resume.
4. Asynchronous versus synchronous.
5. Between versus within instructions.

In general, the first alternative of these pairs is easier to implement and may be handled after the completion of the current instruction.

Precise exceptions

An exception is precise if all the instructions before the exception finish correctly and all those after it do not change the state. Once the exception is handled, the latter instructions are *restarted* from scratch.

Ld.D F4, 0(R2)	*IF*	*D*	*EX*	*M*	*WB*									
Mul.D F0, F4, F6		*IF*	*D*	stall	EX_{M1}	EX_{M2}	EX_{M3}	EX_{M4}	EX_{M5}	*M*	*WB*			
Ld.D F8, 0(R3)			*IF*	stall	*D*	*EX*	*M*	*WB*						
Add.D F6, F4, F6					*IF*	*D*	EX_{A1}	EX_{A2}	EX_{A3}	*M*	*WB*			
Add.D F2, F0, F8						*IF*	*D*	$stall_1$	$stall_2$	EX_{A1}	EX_{A2}	EX_{A3}	*M*	*WB*
S.D F2, 0(R2)							*IF*	$stall_1$	$stall_2$	*D*	*EX*	$stall_3$	*M*	*WB*

Exception at EX_{A1} of Add.D F6, F4, F6: Allow the Mul.D and Ld.D to complete and *flush* the two Add.D and S.D.

Exception at EX_{M5} of Mul.D: The following Ld.D has already completed! $\Rightarrow$ either force in order *WB* or "undo".

Both of the above: *Which one has the higher priority? Why?*

Looking forward

- Prediction on the branches.
- Multiple pipelines in parallel.
- Dynamic scheduling of the instructions by the hardware.

Lecture 3: Branch prediction

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

Branches

There are two important questions.

- When is the target address ready?
- When is the condition resolved?

In a simple RISC processor with simplified branch instructions, we might add more hardware in the decode stage to

- calculate the target address and
- resolve the simple conditions.

Downside: complex tests must be done by separate instructions and the decode stage is even more complicated now.

Delayed branches

Assuming a fast branch, with all the information known by the end of the decode stage, what shall I do with instruction i+1?

Branch	*IF*	*D*	*EX*	*M*	*WB*			
i+1		*IF*	*D*	*EX*	*M*	*WB*		
target			*IF*	*D*	*EX*	*M*	*WB*	
target+1				*IF*	*D*	*EX*	*M*	*WB*

- Flush it and lose one cycle.
- Assume that the effect of the branch is delayed by one cycle and bring something from before the branch to put it in this slot. (Maybe even from after it as long as it is not “harmful”.)

$\Rightarrow$ Delayed branches expose the internal organization to the compiler writer and complicates interrupt handling. Moreover, most processors have complicated branches where the result is not known till a few cycles pass.

Speculation

We have already seen a few ‘solutions’ to branches:

- stall,
- fast branches evaluated completely at the decode stage, and
- delayed branches.

Can we do better by *predicting* the decision and executing *speculatively*?

The speculation game

1. Guess the branch target.
2. Execute the branch to verify the guess.
3. Meanwhile, start execution at the guessed position.

We should attempt to minimize the penalty if our guess is right to almost zero. If our guess is wrong the penalty might be higher.

$\Rightarrow$ How often is our guess correct?

$\Rightarrow$ How can we improve this probability?

$\Rightarrow$ What are the penalties?

Branch prediction

It is one of the heavily researched areas in computer architecture during the 1990s.

Fixed: As an example, a processor may always fetch in-line on true conditional branches.

Static: The strategy varies by opcode type but is predetermined.

Dynamic: The strategy varies according to the program behavior and depends on the history of this branch. $\Rightarrow$ Use up-down saturating counters.

Perfect prediction simply converts the delay for conditional branch into that for unconditional branch (branch taken). *The important goal is the minimization of the branch delay not just a higher prediction accuracy.*

Static options

The prediction to continue in-line is quite easy since we already know the target (it is at $PC + 4$). However, this is not efficient.

Most branches (especially backward) are taken.

- Generate the profile of your target applications.
- Find out the most probable decision for each type of branch.
- Implement the hardware to predict statically based on this information.

Dynamic behavior

- Was this branch taken or not taken in the past iterations?
- How many iterations shall we consider?

The simplest case is to look for the last time only in a *branch history table (BHT)*.

⇒ Use the least significant bits of the PC to index a small table.

⇒ Each entry in the table is one bit indicating if the last time was taken or not. Use this bit as your prediction.

⇒ In case of a misprediction, complement this bit.

Problems of the BHT

The BHT is simple to implement but

- multiple PCs may alias to the same location and
- we may have many mispredictions.

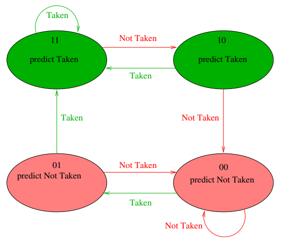
Let us look at an inner loop with four iterations.

Prediction	N	T	T	T	N	T	T	T	N	T	T	T
branch outcome	T	T	T	N	T	T	T	N	T	T	T	N
Misprediction?	+			+	+			+	+			+

We get two misprediction in four decisions.

Improvement: 2-bit prediction

What if we look at the last two iterations? A prediction is not changed untill it misses twice.



The inner loop with four iterations.

Prediction	N	N	T	T	T	T	T	T	T	T	T	T
branch outcome	T	T	T	N	T	T	T	N	T	T	T	N
Misprediction?	+	+		+				+				+

We get only one misprediction in four decisions.

Another 2-bit idea

We can use a saturating up-down two bit counter.

strong Not taken	weak not taken	weak taken	strong Taken
00	01	10	11
N	n	t	T

The inner loop with four iterations once more.

Prediction	n	t	T	T	t	T	T	T	t	T	T	T
branch outcome	T	T	T	N	T	T	T	N	T	T	T	N
Misprediction?	+			+				+				+

We get only one misprediction in four decisions.
This can be easily extended to more than two bits although 2-bit predictors are good enough for many systems.

Branch prediction buffer

This is a table accessed during the fetch cycle by the least significant bits of the PC.

Each entry may be a 2-bit predictor.

If we decode the instruction to be a branch, we have the prediction for it.

This leads to a predicition accuracy in the high 80% to over 99% in some applications.

Correlating predictors

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

- The decision on the third branch depends on the previous two.
- We can correlate the branches!

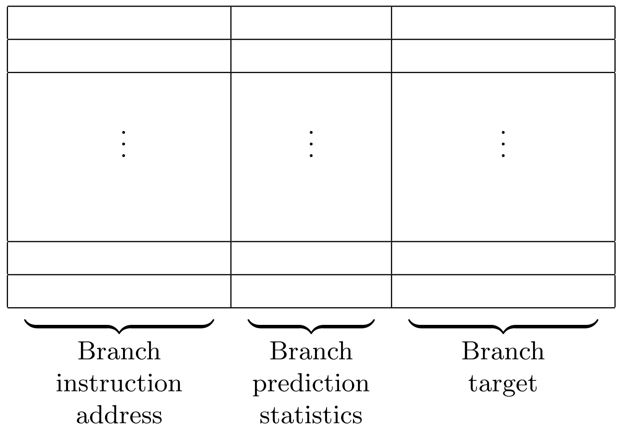
Global history

- A *branch history shift register* keeps the history of the previous few branches. One bit for each branch. For example, for a history of two branches we have two bits and four possible cases: 00, 01, 10, and 11.
- We have a table (of the branch prediction buffer) for each case and decide the prediction on the least significant bits of the PC *and* the global history.

Hybrid schemes

Different predictors work best for different branches.
$\Rightarrow$ Let us combine them and choose the best.

Branch Target Buffer



If the IF “hits” in the BTB, the target instruction that was previously stored in the BTB is now fetched and forwarded to the processor at its regularly scheduled time.

Conclusions

- There are many ways to minimize the effect of control hazards.
- We can achieve very high prediction ratios.
- We must also minimize the penalty of the branch.

Lecture 4: Multiple issue

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

Out of the bottleneck

Let us look at the example of the fire and the well again.

- One person is at the well. There are two lines of people from the well location to the fire. The person at the well fills a bucket and hands it to the next person in one of the lines.
- Two persons are at the well each filling a bucket and then providing it to the lines.

Which one will put the fire down faster? Why?
The “issue rate” of CPI = IPC = 1 is called Flynn bottleneck.

ILP

Instruction Level Parallelsim (ILP) is a property of the software not the hardware. The hardware supports ILP by

- pipelining,
- superscalar in order execution such as in Sun UltraSparc, or
- superscalar out of order execution such as in Intel Pentium4.

The reordering (scheduling) may be dynamic at run time by the hardware or static at compile time by the software.

Going to multiple issue

We will look today at in order execution to solve its problems and detect any dependences. How can we issue two, four, or in general n instructions per cycle?

- Fetch n instructions per cycle,
- decode n instructions per cycle,
- execute n instructions per cycle,
- may access n locations in memory per cycle, and
- may write into n locations in the registers.

Wide fetch

We are not getting the instructions from the real memory but from an instruction cache.

- Instructions are sequential
  - Do they fall on the same ‘line’ in cache? Similar to the issue of aligned and non-aligned accesses to half-words in the memory.
- Instructions are *not* sequential
  - Two serial accesses? No! You will not know the target address and complete the second fetch within one clock cycle.

Solution to wide fetch

Problem: On a taken branch all the fetch slots after the branch are thrown away. ⇒ a low utilization of the fetch unit and eventually a low IPC.

Solution: *Trace cache*

- In addition to the regular cache, store the dynamic instruction sequence.
- Fetch from the trace cache but make sure that the branch directions are correct.
- If you miss get the correct instructions from the regular cache or even from the memory.

A trace cache is used in Pentium4.

Wide decode

Decode: The decoding of a number of instructions

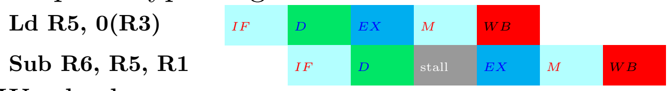
- is easy if they are of fixed length and fixed formats
- but is harder (although possible) for variable length.

Read operands: We should check the dependencies and read the operands.

- With n instructions, we have *at most* $2n$ operands to read in one cycle. ⇒ $2n$ read ports and the register file becomes proportionally slower.

Dependences for n instructions

Remember that we have to stall sometimes even with a complete bypassing network.



We check

$$(s1_{\text{Dec}} = D_{\text{Ex}})\&(op_{\text{Ex}} = \mathbf{Ld}) \quad | \quad (s2_{\text{Dec}} = D_{\text{Ex}})\&(op_{\text{Ex}} = \mathbf{Ld})$$

With two instructions going in the decode, the number of checks quadruples and not just doubles! *n^2 growth in circuits for stall and bypass.*

$$\begin{array}{llcl}
& (s1_{\text{Dec1}} = D_{\text{Ex1}})\&(op_{\text{Ex1}} = \mathbf{Ld}) & | & (s2_{\text{Dec1}} = D_{\text{Ex1}})\&(op_{\text{Ex1}} = \mathbf{Ld}) \\
| & (s1_{\text{Dec1}} = D_{\text{Ex2}})\&(op_{\text{Ex2}} = \mathbf{Ld}) & | & (s2_{\text{Dec1}} = D_{\text{Ex2}})\&(op_{\text{Ex2}} = \mathbf{Ld}) \\
| & (s1_{\text{Dec2}} = D_{\text{Ex1}})\&(op_{\text{Ex1}} = \mathbf{Ld}) & | & (s2_{\text{Dec2}} = D_{\text{Ex1}})\&(op_{\text{Ex1}} = \mathbf{Ld}) \\
| & (s1_{\text{Dec2}} = D_{\text{Ex2}})\&(op_{\text{Ex2}} = \mathbf{Ld}) & | & (s2_{\text{Dec2}} = D_{\text{Ex2}})\&(op_{\text{Ex2}} = \mathbf{Ld})
\end{array}$$

Wide execute

Shall we put n execution units?

- Yes for ALU.
- No for floating point division since it is big and used infrequently.

$\Rightarrow$ based on the instruction statistics, provide a mix of units.

RS/6000:	1 ALU/memory/branch + 1 FP
Pentium II:	1 ALU/FP + 1 ALU + 1 load + 1 store + 1 branch
Alpha 21164:	1 ALU/FP/branch + 2 ALU + 1 load/store

n^2 bypass

The bypass detection logic grows as n^2. This is acceptable since the sources and destinations are small fields (5 bits for 32 registers).
However, the bypass buses also grow as n^2. This is *not* acceptable. The busses are 32 or 64 bits wide each.

- It is difficult to layout and route all of these wires.
- Wide multi-input multiplexers are slow.

$\Rightarrow$ Group functional units into *clusters* and issue the dependent instructions to the same cluster.

Wide memory and write back

There is nothing too special about these two stages for wide issue. Their complexity just grows and they may become slower.

- Additional ports.
- Conflict detection logic for simultaneous multiple reads and writes to the same bank.

Summary

The are some problem spots for in order superscalar processors.

Fetch and branch prediction: may use trace cache.

Decode: the dependence checks grow as n^2.

Execution: Clustering may solve the n^2 bypass buses problem.

Lecture 5: Out of order, Dynamic versus Static scheduling

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

Where are we?

- We pipeline the instructions to enhance the throughput.
- We included long instructions.
- We handled exceptions.
- We managed in order issue to multiple pipelines in parallel.

Now we are ready to have out of order execution.

- Dynamic scheduling in the hardware.
- Static scheduling in the software.

Why do we go out of order?

Div.f F0, F1, F2	IF	D	EX_{D1}	EX_{D2}	$EX_{\dots}$	EX_{D9}	M	WB
Add.f F3, F0, F4		IF	D	stall$_1$	stall$_{\dots}$	stall$_8$	EX_{A1}	EX_{A2}
Mul.f F8, F2, F5			IF	stall$_1$	stall$_{\dots}$	stall$_8$	D	EX_{M1}

- The **Add.f** must wait because of the RAW hazard.
- The **Mul.f** does not need to wait. It can start *before* the **Add.f** instruction.

Dynamic scheduling

Execution: is non-sequential (not the original order)

- This reduces stalls,
- improves the utilization of the functional units, and
- enables parallel execution.

Exceptions: must be precise. We must maintain the *appearance* of sequential execution. This is important but hard.

Instruction buffer

- Instructions are brought from memory and *dispatched* to the decoder in order.
- The decoder must be able to look at multiple instructions to re-order.
- Some instructions will be *issued* (started) while others are waiting.

The instructions reside in the *instruction buffer* while the decoder checks them. (Different names are used by various people.)

Dispatch and Issue

Dispatch: is the first part of decoding.

- The new instructions get a location *in order* in the instruction buffer.
- If the buffer is full, the dispatching unit *stalls all the following instructions*.

Issue: is the second part of decoding

- Start the execution, i.e. send instructions from instruction buffer to execution units *out of order*.
- An instruction that has to *wait does not delay the following instructions*.

Register renaming

```
Div.f   F0, F1 ,F2        Div.f   L0, L1 ,L2
Add.f   F3, F0, F4        Add.f   L3, L0, L4
Mul.f   F0, F1, F2   ⇒    Mul.f   L5, L1, L2
Add.f   F4, F0, F2        Add.f   L6, L5, L2
```

Think of the registers as *names* not specific locations.

On a write, allocate a new location and record it in a map table.

On a read, find the location in the table of the most recent write.

De-allocation occurs when the dependent RAW hazards are cleared.

This is a neat idea that might be implemented in either the hardware or the software. It eliminates WAW and WAR hazards.

Other techniques: Scoreboards and Tomasulo's algorithm

- The scoreboard technique uses a centralized approach to check and resolve the dependencies. It was first implemented in CDC6600 in 1964.
- The Tomasulo data flow technique uses a distributed approach where the reservation station may also contain the *value* of the register not just a tag. This amounts to a register renaming scheme. It was first implemented in IBM 360/91 in 1967.

Those who are interested can read more about these techniques in the "Quantitative Approach".

Very Long Instruction Word

We have seen some problems with multiple issue superscalars:

- N^2 dependence checks (large stall and bypass logic),
- N^2 bypass buses (partially fixed with clustering), as well as
- wider fetch and problems with branch prediction.

In VLIW,

- a single issue pipeline that has N parallel units is used,
- the compiler only puts independent "instructions" in the same group,
- VLIW travels down the pipeline as one unit, and
- in *pure* VLIW machines the processor does not need to do any dependence checks.

VLIW purity

In a pure (classical/ideal) VLIW design the compiler schedules the pipeline including the stall cycles.

$\Rightarrow$ The compiler must know the exact latencies and organization of the pipeline.

Problem 1: These details vary in different implementations. We must recompile the code. (TransMeta recompiles on the fly.)

Problem 2: Even for a specific implementation, the latencies are not fixed. What shall the hardware do for a cache miss?

Real implementations are not ideal.

Scheduling and issuing

Schedule: Decide the *order* of the instructions.

- Put independent instructions between the slow operations and the instructions that need their results.

Issue: Decide the *time* a specific instruction starts.

- Once all the dependencies are clear we can start.

	Schedule	Issue
Pure VLIW	SW	SW
In-order superscalar	SW	HW
Out-of-order (dynamic)	HW	HW

Scheduling: Compiler or HW

Compiler:

- + Large scope (may be the whole program).
- + Leads to a simpler hardware.
- – Low branch prediction accuracy.
- – No information about memory delays (cache misses).
- – Difficult to speculate and recover.

Hardware:

- + Better branch prediction accuracy.
- + Dynamic information about memory delays.
- + Easier to speculate and recover.
- – Finite resources to buffer instructions.
- – Complicated hardware (harder to verify, may lead to slower clock).

Compiler techniques

We want to increase the number of independent instructions.

Loop unrolling: Put more than one iteration in sequence in a wider loop.

Software pipelining: Similar to what happens in hardware, a part of the first iteration is done with a part of the second iteration.

Trace scheduling: Programs include other things beyond loops.

Those who are interested can read more about these techniques in the “Quantitative Approach”.

Where are we now?

- Pipelines.
- Exceptions.
- Muliple issue.
- Dynamic scheduling.
- Static scheduling.

Next we go to the memory system.

Lecture 6: Introduction to memories

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

Inside large digital integrated circuits

Memories
- for temporary storage of results (registers),
- for the reduction of the information retrieval time (caches),
- or as the main store of information (main memory, virtual memory).

Control logic blocks handle the flow of information and assure that the circuit performs what is desired by the user.

Datapath blocks
- the real engine that performs the work.
- Mainly perform either some arithmetic or logic operations on the data.

Communications between all the elements is via wires usually arranged in the form of buses.

Classification: write, read, and erase

- Some technologies allow only a single writing and many reads, others allow re-writing.
- In some technologies a new writing overrides the older information, other technologies need an erase cycle.

How do you classify: clay, pottery, paper (pencils and pens), and classboards?

Classification: sequential versus random access

Random access means that we access any location (that you choose randomly) in the same amount of time.

- *Does the tape provide random access?*
- *Is the ROM a random access memory?*

Think of turning your eyes in a large room versus moving with your legs.

Classification: volatility

Do we loose the information when we turn the power off?

Non-volatile: such as ROM, Flash, CD, . . .

Volatile: such as RAMs.

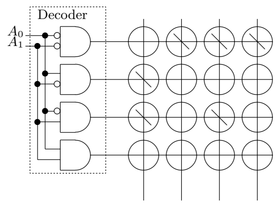
Contrast pottery to a statement written in the sand on the beach.

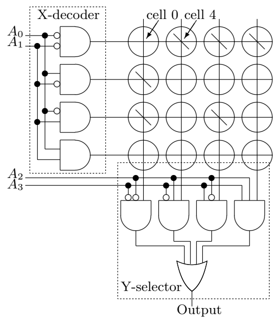
Access time and capacity

- Let us order the memories within a computer according to the time taken to retrieve the information.
- Now, let us do it according to the capacity.

Memory chip organization

The size of the memory grows exponentially with the number of bits in the address. Capacity = $\underbrace{2^{\text{address lines}} \times \text{data lines}}_{\text{Organization}}$.





What is the effect of the number of address lines and data lines on the speed? Why?

Inside the cell

The design of the cells leads to different memory technologies.

Wire	ROM
Fuse	PROM
Floating gate	EPROM, EEPROM, Flash
Latch	Static RAM
1T+1C	Dynamic RAM

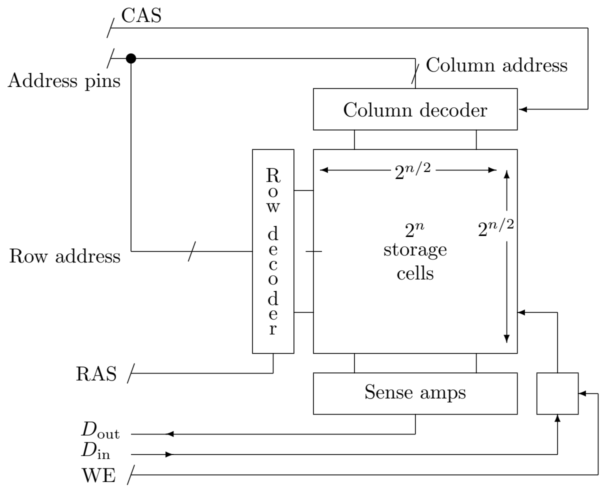
SRAM and DRAM

	Static RAM	Dynamic RAM
Cell composition	many transistors	1 T + 1 C
Cell size	bigger	smaller
Density	lower	higher
Capacity	smaller	larger
Speed	faster	slower
Power consumption	more	less
Retention	as long as power is on	needs refreshing

Which one do we use in caches? And in the main memory?

Back to DRAM chips

DRAM addressing is divided to rows and columns.



DRAM enhancements

The DRAM controller multiplexes the address and provides the RAS and CAS to the chip. To get a faster access, we use

- a fast page mode,
- SDRAM (S for synchronous) or DDR SDRAMs, or
- RDRAMs (R for Rambus).

Memory summary

A memory is an entity that holds the information for a later use.

- A memory with a small capacity fetches the information faster than another with a larger capacity. *Why?*
- A memory with a small capacity may use larger cells with more power to provide an even faster operation. *Why does a bigger cell and more power translate to speed?*
- Decoding is in levels. Remember your own algorithm to reach the room 8208.

Lecture 7: Caches

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

Introduction to caches

- Why do we have memories in computers?
- What is the hierarchy of storage elements (latency, bandwidth, capacity, and price)?
- Why is there a difference in speed?

Imagine yourself in a large library with many books. You want to read a number of sections from a few books. What are you going to do?

Why do we have caches?

On the average, our goal is to give the processor the illusion of a *large* memory system with a *short* effective access time.

```
for(i=0;i<n;i++)
    sumsq  = sumsq + x[i]*x[i] + y[i]*y[i];
```

The basic principles of locality:

1. Spatial locality.
   - Sequential access.
2. Temporal locality.

Some definitions

When the processor fetches a piece of information it might be an instruction or a data value. Hence,

- we may use a unified (integrated) cache for both or
- we may use a split I$ and D$.

We may also have multiple levels of caches. The processor tries first to find the information in the nearest (highest) level of the hierarchy.

- The information request may *hit* in the cache and the needed word reaches the processor after the *hit time* or
- it may *miss* in the cache and is retrieved from the lower level of the hierarchy after an additional *miss penalty*. (*miss time* = hit time + miss penalty)

How much time are we loosing on misses?

- Each instruction accesses the memory for its own fetch.
- It may also access the memory for data.

$$\Rightarrow \text{memory accesses per instruction} \geq 1.$$

Each instruction may hit or miss. If we profile our applications we get the *hit rate* and *miss rate* and calculate

$$\begin{aligned} AMAT &= \text{hit time} \times \text{hit rate} + \text{miss time} \times \text{miss rate} \\ &= \text{hit time} + \text{miss rate} \times \text{miss penalty}. \end{aligned}$$

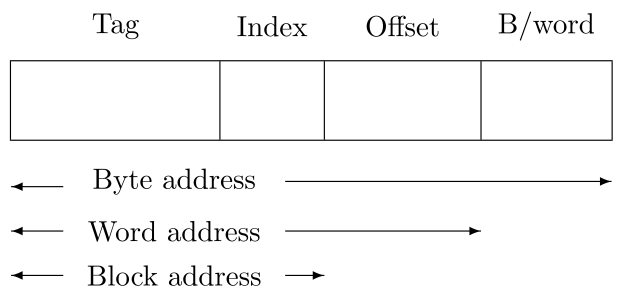
Implementation

To understand how caches work, let us ask a few fundamental questions. Here are the first two.

1. Where is the block placed in the cache?
   - Simplest is *Index = (Block address)mod (# blocks in cache)*. This is called direct mapping.
2. Is the block available (hit) in the cache?
   - Each block in the cache is associated with a tag (and a valid bit). If the requested block has the same index but a different tag it is a miss.

Let us think about a cache with 8 blocks, each one word, that is initially empty and the references: 22, 26, 22, 26, 16, 4, 16, and 18 to words in the memory.

Address



Let us try to find the size of a cache with 9 bits index in a machine having 32 bits for its addresses assuming that the word is four bytes and the block is eight words. *Why eight words in a block?*

Back to fundamentals

Here is another fundamental question

3. In a miss and a need to replace a block, which one shall I choose?
   - Trivial for direct mapping.
   - Random, Least recently used, or FIFO for other mapping techniques that we will study later. *Why do we need other mapping techniques?*

Write policies

The last fundamental question is

4. what happens on write?

Write through:	Write to the lower level as well. May slow things down. $\Rightarrow$ use a write buffer.
Write back:	(or Copy back) write only when the block is replaced. $\Rightarrow$ minimize the traffic by indicating if the block is *dirty*.

What about a miss at the time of writing (remember the case of multiple words per block)?

Write through	Copy back
Write allocate	$\boxed{\text{Write allocate}}$
$\boxed{\text{No-write allocate}}$	No-write allocate

Now what

A designer seeks to reduce

- the miss penalty,
- the miss rate, and
- the hit time.

We must balance that with the rest of the hierarchy as well.

Lecture 8: Faster Memory

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

Access time and bandwidth

Let us look at a large system: many processors each with its own cache accessing the memory system.

The access time of the memory is the period from the time the address is given to the memory chip till the data is ready to be sent to the requester.

The bandwidth is the number of bytes (or requests) served per unit time.

The bandwidth is not simply the inverse of the access time.
Why? How can you increase the bandwidth given a fixed access time?

Higher bandwidth: wide memories

Example 1 In a system, the access time of main memory is 10 clock cycles. The cache block size is 16 bytes. The address is transfered on the bus in one clock cycle and any result from the memory is transfered back in one cycle. What is the bandwidth for a narrow bus and memory (4 bytes) and that of a wide bus and memory (16 bytes)?
Solution: We may assume that the address of the whole block is sent only once and that the memory results are not overlapped with the memory access.

$$\begin{aligned} T_n &= 1 + 4 \times (10 + 1) = 45 \text{ cycles.} \\ BW_n &= 16/45 \approx 0.3556 \text{ bytes/cycle} \\ T_w &= 1 + 10 + 1 = 12 \text{ cycles.} \\ BW_w &= 16/12 \approx 1.3333 \text{ bytes/cycle} \end{aligned}$$

Is it worth the price?

Higher bandwidth: interleaved memories

What about multiple memory banks but a single narrow bus?

Example 2 Now, if we use a simple interleaving scheme where four banks are used and each clock cycle one of them starts to access its data, what is the bandwidth?
Solution: In this case, at time 0 the address is sent then at time 1 the first bank starts and its data is ready at time 11. The second bank starts at time 2 and its data is ready at time 12 and so on.

$$\begin{aligned} T_i &= 1 + 10 + 4 \times 1 = 15 \text{ cycles.} \\ BW_i &= 16/15 \approx 1.0667 \text{ bytes/cycle} \end{aligned}$$

In word interleaved systems, the bank number is

$$(\text{word address}) \bmod (\# \text{ banks}).$$

How much time are we loosing on misses?

- Each instruction accesses the memory for its own fetch.
- It may also access the memory for data.

$$\Rightarrow \text{memory accesses per instruction} \geq 1.$$

In addition to AMAT, we calculate the total CPU time for the program.

$$\begin{aligned} \text{CPU time} &= (\text{CPU cycles} + \text{Stall cycles}) \times \text{Clock cycle} \\ &= (\text{CPU cycles} + \text{Read Stalls} + \text{Write Stalls}) \times \text{Clock cycle} \\ \text{CPU time} &\approx IC \times (CPI + \frac{Mem\ access}{Inst.} \times \text{miss rate} \times \text{miss penalty}) \\ &\quad \times \text{Clock cycle} \end{aligned}$$

Simple example

Example 3 A system with CPI of 1 has load/store frequency of 25% with cache miss rate 5% and miss penalty 10 cycles. A suggested change to the cache reduces the miss rate to 2% but increases the size of the clock cycle by 20%. Should you use this change?
Solution: We should calculate the CPU time for both cases to decide

$$\begin{aligned} T_1 &= IC(1 + (1 + \frac{25}{100}) \times 0.05 \times 10) \times cycle \\ T_2 &= IC(1 + (1 + \frac{25}{100}) \times 0.02 \times 10) \times 1.2cycle \\ \frac{T_1}{T_2} &= \frac{1 + 1.25 \times 0.5}{(1 + 1.25 \times 0.2) \times 1.2} = \frac{1.625}{1.5} \end{aligned}$$

Yes, this change is beneficial.

Is 'faster' really that much better?

Example 4 The gcc compiler runs on a system with a miss rate of 5% for instructions and 10% for data. The perfect CPI is 1 cycle, the miss penalty is 12 cycles, and the load/store frequency is 33%. Study the effect of changing to 1) a faster architecture with $CPI = 0.5$ instead of 1 and 2) a faster system with three times the clock frequency.
Solution: First let us calculate the original CPU time

$$\begin{aligned} T_{orig} &= IC(1 + 0.05 \times 12 + \frac{1}{3} \times 0.10 \times 12) \times cycle \\ &= IC(1 + 1) \times cycle \end{aligned}$$

Is ‘faster’ really that much better?

The changed CPU times are

$$\begin{aligned} T_{CPI} &= IC(0.5 + 0.05 \times 12 + \frac{1}{3} \times 0.10 \times 12) \times cycle \\ &= IC(0.5 + 1) \times cycle \\ T_{freq} &= IC(1 + 0.05 \times 36 + \frac{1}{3} \times 0.10 \times 36) \times \frac{1}{3} cycle \\ &= IC(1 + 3) \times \frac{1}{3} cycle \end{aligned}$$

The smaller CPI gives only $1.5/2 = 3/4$ reduction while the higher frequency gives only $(4/3)/2 = 2/3$.
Remember that marketing is different from engineering!

Final notes

- The hit time is also important.
- Spatial locality leads us to blocks with multiple words.
- A larger block size may increase the miss penalty. We must balance different factors.
- Wide and interleaved memories boost the system performance.
- The total time taken by a program is a very important measure.

Lecture 9: Virtual Memory

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

Virtual memory

Caches and main memory give the processor the illusion of a large and fast memory.
A second level of the hierarchy is between the main memory and hard disk. This level gives the illusion of a much larger memory at a reasonable speed.
We will look at the same set of questions.

1. Where to put the "block" (*page*)?
2. How to find it?
3. What if it is not found (*page fault*)?
4. What is the writing policy?

Why?

A much larger memory helps in two main aspects.

1. Now, a single program may be larger than the physical real memory.
2. The programs use a *virtual address* but is located in the real memory at a *physical address*. This virtual to physical translation helps in multiprogramming.

Relocation:	A process is loaded in any physical frame.
Protection:	Each process is isolated from other processes and cannot access their physical addresses.

Disk technology

- Disks have cylinders, tracks, and sectors.
- Outer tracks are physically longer than inner ones.
  - Old disks used the same data size for all sectors.
  - New disks use a constant linear recording density and put more data on outer tracks.

Disk times

Seek time: Time for the arm to reach the track. Disk companies publish the maximum and minimum seek time between cylinders. *Their average is not what you should use!*

Rotational time: Time for the disk to rotate and get the required sector under the head. The average is usually taken as half of the time for a full rotation. (0.5/RPM)

Transfer time: Time for the disk to read the data and provide it to the controller. (sector size/transfer rate)

Controller overhead: Time taken to connect to the bus and resolve any queuing delays.

Implications of slow disks

- Direct mapping leads to conflict misses although we might have other empty slots. Can we use a better mapping to minimize the page fault rate?
- Disks transfers are in sectors, we should not think about individual words. Page sizes are typically between 4 kB and 16 kB.

Mapping the blocks

Instead of a high conflict miss due to direct mapping (the block goes to exactly one location) we can map the block to a set of locations. Within the set, the block may be placed randomly.

$$Index = (Block\ address) mod\ (\#\ sets\ in\ cache)$$

For a cache that may host n blocks

- a direct mapped cache has n sets,
- a two way set associative has $n/2$ sets (each set has two locations),
- a four way set associative has $n/4$ sets (each set has four locations), and
- a fully associative cache has one set where any block may be placed in any location.

For virtual memory, we use fully associative mapping.

Address translation

$$\begin{tikzcd}
\boxed{\text{Virtual page number}} \arrow[d] & \boxed{\text{page offset}} \arrow[d] \\
\boxed{\text{Physical page number}} & \boxed{\text{page offset}}
\end{tikzcd}$$

This translation is done by a page table located in memory. *Does that mean that we access the memory twice for each request once for the translation and the second for the actual transfer?*

The page table

It is a table addressed by the virtual address and containing the physical address. *Do we need tags? Do we need a valid bit?*

- If the virtual address is 32 bits, the physical address is 32 bits, and the page size is 4 kB, how many pages can we have?
- If each entry in the page table is 4 bytes, what is the expected size of the table?
- If the page is not in the physical memory its disk address may be recorded in the page table or in a separate structure maintained by the OS.

Page table size limitation

- A single limit register.
- If the programming memory model has two segments (heap and stack) then maybe use two limit registers.
- An inverted page table has a number of entries equal to the number of physical pages only.
- Can we page the the page table?

Replacement

In direct mapping the replacement is trivial. In fully associative, we must think about it.

- First In First Out.
- Least Recently Used.
- Random.

Do we need any extra bits to implement LRU replacement?

Write policy

Since disks are much slower than semiconductor memories, we use write back (also called copy back) and not write through. We must use an additional *dirty bit* to know if the page was changed while in memory or not.

Summary

- Naming convention

caches	virtual memory
block (a few words)	page (4kB to 16kB)
miss	fault

- Policies are heavily influenced by the much slower disk.

Page placement	fully associative
Page identification	via translation by the OS
Replacement	LRU (sophisticated)
Write strategy	write back with write allocate

Lecture 10: Caching the page table

Hossam A. H. Fahmy

Cairo University, Faculty of Engineering

Overview

Translation look-aside buffer (TLB)

- Spatial locality means that the program will be accessing the same page many times. There is no need to retranslate.
- Temporal locality means that the program will be accessing the pages that were recently accessed.

$\Rightarrow$ Use a small storage within the processor to buffer the recent translations. This TLB is effectively a type of ‘cache’ for the page table.

TLB design

- Each entry has the virtual page number and the physical page number as well as the reference and dirty bits of the page.
- The block size is one or two page table entries and the TLB size is between 32 and 1024 blocks.
- A fully associative mapping is used with a random replacement policy. *Why not LRU?*
- A write back policy is used. *The only part that might change is the reference and dirty bits.*

Sequence of events

1. The processor generates the virtual address.
2. It is checked in the TLB.
   - In case of a TLB hit, the virtual address is translated.
   - For a miss, the TLB is updated and the translation completed.
     - If the page is not in physical memory, then allocate it.
3. The physical address goes to the cache.
   - In case of a cache hit, the data is retrieved and sent to the processor.
   - In a miss, the data is retrieved from the main memory.

Can we make things faster?

- A *virtually addressed* cache may lead to aliasing when two programs attempt to access the same location (as in printing a file) with two different virtual addresses.
- If the whole cache index falls within the bits of the page offset then the access to both the cache and TLB can occur in parallel.

Protection

User programs must be protected from any process that tries to access their private data.

- The OS maps the virtual address space of each process to a different set of physical pages.
- The user programs cannot alter the page table.
- The sharing of data between processes is controlled by the OS including the permissions for reading and writing.

Requirements for protection

The minimum requirements are

1. two operation modes: a user mode and a system mode,
2. the CPU must allow the user program to request a service from the system (this switches the mode bit in the CPU state to the system mode),
   and
3. some instructions are *privileged* and are only allowed in system mode.

In the case of context switching, we either invalidate all the data in the TLB or we include the process ID in the entries and allow each process to access its own entries only.

Page faults are exceptions

Page faults must be precise exceptions.

1. All the instructions before the exception finish correctly.
2. All the instructions after it do not change the state.
3. Once the exception is handled, the latter instructions are *restarted* from scratch.

The CPU saves the cause the of the exception (page fault) and the the program counter value of the instruction that must be restarted then switches to the system mode.

Handling the page fault

Because it is a page fault, the OS will

- find the location of the referenced page on the disk,
- choose a physical page to replace (if it is dirty, write it to the disk),
- start a read to bring the needed page from the disk,
- save the whole state of the current process, and
- start another process while the disk is bringing the page.

Misses

Four C's cause misses in caches. These same types affect the virtual memory and the TLB as well.

Compulsory misses: The first time we reference a word.
- reduce it by using multiple words in the block.

Capacity misses: No free space in the cache.
- reduce it by using a larger cache.

Conflict misses: Multiple blocks map to the same location.
- reduce it by using a higher associativity.

Coherence misses: Another system component (another processor or I/O) invalidates the cache location.
- reduce it by limiting parallelism!

The miss rate reduction techniques just mentioned may increase the hit time and increase the cost.

Summary

- Caches
  - Split versus integrated
  - Direct, set associative, and fully associative.
  - Replacement and write policies.
- Main memory
  - Wider memories and interleaving for higher bandwidth.
  - Virtual memory for larger programs and multiprogramming.
- TLB
  - To quickly implement the virtual to physical translation.