Advanced Computer Architecture

Chapter 2: CPU Architecture

Lecturer: *Paul H J Kelly*

Autumn 2001

Department of Computing
Imperial College

Room 423
email: phjk@doc.ic.ac.uk

Course web site:
http://www.doc.ic.ac.uk/~phjk/AdvancedComputerArchitecture.html

CHAPTER 2

Most students should have seen much of this before...

Alternative textbook: Patterson and Hennessy's *Computer Organisation and Design, the Hardware and Software Interface* has a detailed intro to pipelining (Ch.6) (but does *not* help with the rest of the course).

Textbooks:

- Hennessy and Patterson, *Computer Architecture Architecture: a quantitative approach, second edition* Morgan Kaufmann 1996. £29.95.

  > "Once in a great while, a landmark computer-science book is published. Computer Architecture: A Quantitative Approach, Second Edition, is such a book. In an era of fluff computer books that are, quite properly, remaindered within weeks of publication, this book will stand the test of time, becoming lovingly dog-eared in the hands of anyone who designs computers or has concerns about the performance of computer programs." — Robert Bernecky, Dr. Dobb's Journal, April 1998

2.1 Elementary Processor Architecture

A computer is a machine for executing instructions. We will begin by considering a straightforward instruction set:

- 32 general-purpose registers: `R`*i*
- Load/store instructions:
  - `LW Ri,` *address*
  - `SW` *address*`, R`*i*
- Register-register arithmetic:
  - `ADD R`*i* `R`*j* `R`*k* (meaning `R1:=R2+R3`)
- Unconditional and conditional jumps:
  - `J` *address*
  - `BEQZ R`*i*`,` *address*
    (meaning if `R`*i*=0 then `PC :=` *address*)

(We will use the DLX architecture from H & P)

How is an instruction executed? This instruction set can be handled in a five step cycle (H&P pp.127):

IF instruction fetch

Program counter sent to main memory

ID instruction decode

Instruction received from main memory. Registers used in instruction are accessed.

EX execution

Arithmetic and logical operations are performed, including calculation of effective addresses for loads and stores.

MEM memory reference

The memory access for load and store instructions takes place.

WB write back

Write the result to the register specified (either from the ALU or from the memory)

In more detail:

EX
- In arithmetic instructions, this is when the calculation is done
- In load or store instructions, the effective address is calculated
- In conditional branches, this is the natural point for the comparison
- For computed jumps, this is where the branch target address is calculated

MEM
- In a store, the calculated address and source register are sent to memory.
- In a load, the calculated is sent to memory
- In a branch, the PC is set
- For arithmetic operations, this stage is inactive

WB
- For arithmetic and load instructions, write the result to the register specified
- For branch, this stage is inactive

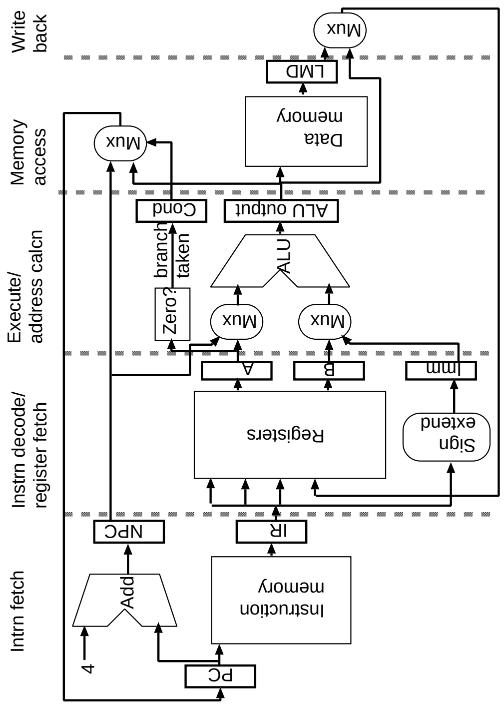

Figure 1: Data path for non-pipelined DLX processor

- Thus, the execution of every instruction follows the sequence

  IF ID EX MEM WB

  (although not every stage is active in processing every kind of instruction)

- In a straightforward, sequential implementation, some instructions can be completed in less than five steps, by skipping inactive stages
- The PC is written in the MEM stage
- The registers are written in the WB stage
- Backward arrows cause trouble later...

PERFORMANCE OF A SEQUENTIAL PROCESSOR

The performance of a simple, sequential machine is determined by the clock cycle time, and the average number of clock cycles per instruction.

- The clock cycle time is dependent on the implementation technology - e.g. logic family (CMOS, ECL), fabrication geometry (e.g. 0.5μm, 0.1μm), semiconductor material (e.g. silicon, gallium arsenide, indium phosphide?).
- It also depends on the complexity of the most complex operation which has to happen within a single cycle.

CPI

- The number of clock cycles needed depends on the particular instruction
- The execution time of a program is the sum of the execution times of its instructions
- A common figure of merit for an architecture is CPI, the average number of clock cycles per instruction
- CPI depends on the program being executed
- Reducing CPI is a good idea
- But reducing average CPI might not improve performance

Reducing average CPI might not improve performance?

- A better instruction set might enable much useful work to be done using a single instruction
- A better compiler might make better use of such long-running instructions
- By breaking instruction processing down into more steps, it might be possible to simplify each stage — so that the clock can run faster

2.2 Pipelining

Execution of an individual instruction is inherently a process consisting of several steps, each depending on the previous one.

How can we reduce CPI?

- Design the instruction set so few steps are needed.
- Improve the compiler so faster instructions are used in preference to slower ones.
- Start on the next instruction before the preceding one has finished

$\hookrightarrow$ that is, *pipelining*

The basic idea :

Inst^n									
1	IF	ID	EX	MEM	WB				
2		IF	ID	EX	MEM	WB			
3			IF	ID	EX	MEM	WB		
4				IF	ID	EX	MEM	WB	
5					IF	ID	EX	MEM	WB
Cycle	1	2	3	4	5	6	7	8	9

On each clock cycle another instruction is fetched and begins its five-step execution. Each instruction is completed five cycles later.

The CPI is now 1.

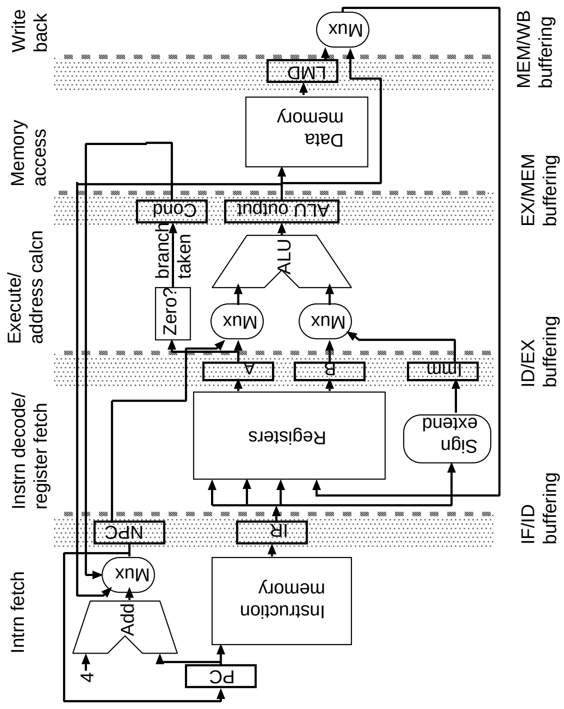

Figure 2: Data path for pipelined DLX processor

- Pipelining is implemented by adding buffering registers between each pair of pipeline stages
- The New PC multiplexor has been moved to the IF stage so the PC is only updated in IF
- Two reverse flows: PC and register values

  Reverse flows lead to complications.

QUESTION 1 : How much does pipelining cost?

QUESTION 2 : What might go wrong?

COSTS OF PIPELINING

(H&P pg 135)

This performance improvement is not free.

- If a unit in the data path is used in more than one pipeline stage, it must be replicated (or contention would occur)
  - (Example: since the ALU is used in EX, the PC must be incremented (in IF) by a separate increment)
  - (If the instruction set were to include instructions in which several ALU operations are involved, we would have to have several ALUs)

(costs of pipelining, continued, 1...)

- The memory system must now handle much more traffic (H&P pp.142):
  - **BEFORE:**
    one instruction fetch (in IF) + one read or write (in MEM) per 5 cycles
  - **AFTER:** one instruction fetch + one read of write per cycle
- Note that even in the sequential case, we assumed that the memory access latency was one cycle - so using the memory every cycle just uses spare capacity.
- A common way to deal with this is to introduce a separate memory for instructions.
- In practice, instead of two memory systems. We just have two caches, I-cache and D-cache.

(costs of pipelining, continued, 2...)

- The cycle time may be increased because of the increased size and complexity of the processor, and because of the need to buffer between stages.

  This overhead on introducing additional pipeline stages limits the speedup available through pipelining.
- The control unit will be bigger and more complicated.

2.3 What might go wrong: Hazards

A hazard occurs when a stage in processing an instruction cannot complete for some reason. The instruction is then stalled for one of more clock cycles.

WHY?

- Structural hazards

  The stalled stage required some hardware which is currently in use.
- Data hazards

  The stalled stage requires some data which is no yet available, or would destroy some data yet to be used.
- Control hazards

  A branch changed the PC so the next instruction is not yet known.

(see H&P pp.146)

- Structural hazards
  - A good example of a structural hazard occurs in machine with only one memory port.
  - The instruction fetch stage needs to read from memory in every cycle.
  - The memory access stage needs to read or write in those cycles where it is processing a load or store instruction.
  - Thus, the IF stage is stalled if the MEM stage is using the memory port.
  - Therefore we lose a clock cycle for every load or store instruction executed.

- Data hazards

  Data hazards occur when the order of access to operands is changed by the introduction of pipelining.

Example:

```
ADD R1, R2, R3    (R1:=R2+R3)
SUB R4, R1, R5    (R4:=R1-R5)
```

Inst^n						
ADD R1,R2,R3	IF	ID	EX	MEM	WB	
SUB R4,R1,R5		IF	ID	EX	MEM	WB
Cycle	1	2	3	4	5	6
					↑R1 written here	
			↑R1 read here			

- Unless we do something about it, the subtraction will use the wrong data.

Dealing with data hazards—

- **IDEA:** stall the subtraction until the WB stage of the addition completes.
- **SMARTER IDEA:** "forwarding" – check whether source register is same as last destination register.

  If so, route ALU's output direct to where it's needed.

We return to this in more detail very shortly.

(See H&P pp.147)

- **Control hazards**

  A control hazard is like a data hazard, but worse. The new PC may not be known until MEM, but the IF stage of the following instruction must know its value.

  Unfortunately, the forwarding/bypass trick is inapplicable.

  There are several different ways to address the problem:

  – delayed branch - redefined instruction set so branch takes effect one instruction later

  – predict branch taken

  – predict branch not taken

  – dynamic branch prediction

(See H&P pp.161 and pp.262)

2.4 Pipelining: avoiding stalls

In this course we will focus on the role of parallelism in computer architecture. We will not, therefore, consider certain important topics further, for example :

- Interrupts and exceptions (see H&P pp.179)
- Virtual address translation and page faults (see H&P pp.445)

We will concentrate on making maximum use of parallelism in processing arithmetic instructions....

Bypassing to avoid stalls

Recall the example:

```
ADD R1, R2, R3    (R1:=R2+R3)
SUB R4, R1, R5    (R4:=R1-R5)
```

Inst^n						
`ADD R1,R2,R3`	IF	ID	EX	MEM	WB	
`SUB R4,R1,R5`		IF	ID	EX	MEM	WB
Cycle	1	2	3	4	5	6
					↑R1 written here	
			↑R1 read here			

- Because of pipelining, sequence of events no longer coincides with the data dependence.
- However, if we examine this example more carefully, it is apparent that the value is in fact calculated in time - it just isn't in the register yet.

(H&P pp.147)

- The idea of bypassing (often called forwarding) is to add special hardware to recognise this hazard, and to route the value direct to where it is needed.

OBSERVE:

Inst^n						
`ADD R1,R2,R3`	IF	ID	EX	MEM	WB	
			↑R2+R3 calculated here			
`SUB R4,R1,R5`		IF	ID	EX	MEM	WB
				↑R2+R3 needed here		
Cycle	1	2	3	4	5	6

- So all we have to do is copy the ALU output back to the ALU input ready for the next cycle.
- Of course we must still make sure R2 + R3 is eventually stored in R1, in case it is referred to again.

HARDWARE REQUIREMENTS FOR BYPASSING

Two parts : data path changes and control.

- **Data Path:**

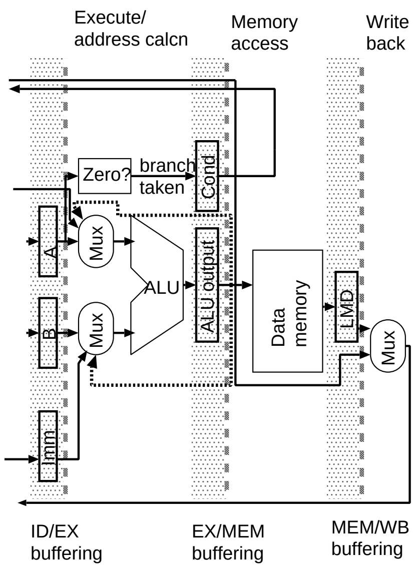

Figure 3: Data path change for forwarding from EX to EX

Hardware requirements for bypassing...

- **Data path:**

  Require additional wiring for bypass paths, and multiplexors to select bypassed or register-file operand.

- **Control:**
  - The bypass multiplexer should be enabled when the result register of instruction$_i$ is the same as the (left or right respectively) operand register of instruction$_{i+1}$ - and instruction$_i$ is an arithmetic instruction.
  - Require additional comparator, to check during instruction$_{i+1}$'s ID stage whether above condition holds.

Bypassing multiple stages

	1	2	3	4	5	6	7	8	9
ADD R1,R2,R3	IF	ID	EX↘	MEM	WB				
SUB R4,R1,R5		IF	ID	$\boxed{\text{EX}}$	MEM	WB			
AND R6,R1,R7			IF	ID	$\boxed{\text{EX}}$	MEM	WB		
OR R8,R1,R9				IF	ID	$\boxed{\text{EX}}$	MEM	WB	
XOR R10,R1,R11					IF	ID	EX	MEM	WB
Cycle	1	2	3	4	5	6	7	8	9
				↑Forward to ADD as before					
					↑Forward to AND				
						↑No forwarding			

- The AND instruction would stall unless the result of the ADD is forwarded — but after one cycle delay
- The XOR needs no forwarding because its ID is after the ADD's WB has completed
- In the H&P DLX design, the OR requires no forwarding: the ADD's WB completes in the first half of the cycle, the OR's ID takes place in the second half.

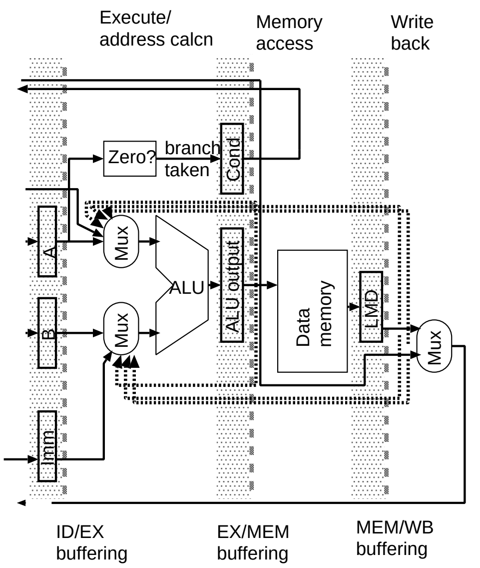

Figure 4: Data path bypassing multiple stages

- Additional path from ALU output via buffer back to ALU inputs
- Additional path also needed from memory output back to ALU input

BYPASS TO MULTIPLE DESTINATIONS

`ADD R1, R2, R3` R1:=R2 + R3
SW 25(R1), R1 Store[25+R1]:=R1

	1	2	3	4	5	6
ADD R1,R2,R3	IF	ID	EX↘	MEM	WB	
SW 25(R1),R1		IF	ID	$\boxed{\text{EX}}$	$\boxed{\text{MEM}}$	WB
				↑R2+R3 used for address		
					↑R2+R3 sent to memory	

- Notice here we have to bypass the register file to route a value direct from the ALU to the ALU as before - but also, we need to use bypassing to route data to the memory address register.
- In general, bypassing may be applied to any pair of functional units.

UNAVOIDABLE STALLS

EXAMPLE:

	1	2	3	4	5	6	7	8	9
LW R1,32(R6)	IF	ID	EX	MEM↘	WB				
ADD R4,R1,R7		IF	ID	$\boxed{\text{EX}}$	MEM.....				
SUB R5,R1,R8			IF	ID	$\boxed{\text{EX}}$......				
AND R6,R1,R7				IF	ID				
Cycle	1	2	3	4	5	6	7	8	9
				↑ADD not OK					
					↑SUB OK				

It is not possible for the ADD to proceed since it needs the value being read from memory at the beginning of the MEM cycle which will fetch it.

A stall is therefore unavoidable. We need to implement a pipeline interlock mechanism:

- **CONTROL**

  Hazard detection is the same as with forwarding : during ID we check whether the register being read is the same as one which will be written by some earlier instruction.

- **TIMING**

  The instruction is delayed until the value it needs is generated :

LW R1,32(R6)	IF	ID	EX	MEM	WB				
ADD R4,R1,R7		IF	ID	*stall*	EX	MEM	WB		
SUB R5,R1,R8			IF	*stall*	ID	EX	MEM	WB	
AND R6,R1,R7				*stall*	IF	ID	EX	MEM	WB
Cycle	1	2	3	4	5	6	7	8	9

2.5 Avoiding Stalls By Scheduling Instructions

Consider the common high-level language statement

```
a := b + c
```

We get

LW R1,B	IF	ID	EX	MEM	WB			
LW R2,C		IF	ID	EX	MEM	WB		
ADD R3,R1,R2			IF	ID	*stall*	EX	MEM	WB
SW, A,R3				IF	*stall*	ID	EX	MEM WB

The stall seems inevitable. However, consider the sequence

```
a := b + c
d := e - f
```


A longer sequence offers more scope:

```
a := b + c
d := e - f
```

Now we can do somewhat better.

```
LW      Rb, B           LW      Rb, b
LW      Rc, c           LW      Rc, c
ADD     Ra, Rb, Rc      LW      Re, e
SW      a   Ra          ADD     Ra, Rb, Rc
LW      Re, e           LW      Rf, f
LW      Rf, f           SW      a, Ra
SUB     Rd, Re, Rf      SUB     Rd, Re, Rf
SW      d, Rd           SW      d, Rd

two stalls              no stalls
```

• Notice that for this to work we need to have plenty of registers (how many extra?)

SUMMARY

- CPI - clock cycles per instruction
- Pipelining: start next instruction before previous has finished
- Faster: potential speedup $\times N$ for N-stage pipeline
- Efficient: less than N-fold increase in hardware
- Hazards: structural, data, control
- Data hazards simplified by instruction set design (see H&P Ch.2)
- Data stalls reduced by forwarding and instruction scheduling