

Computer Systems Architecture

<http://cs.nott.ac.uk/~txa/g51csa/>

Thorsten Altenkirch and Liyang Hu

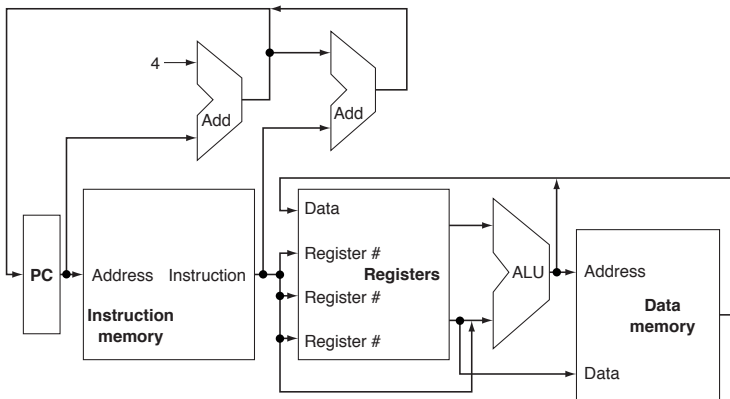
School of Computer Science
University of Nottingham

Lecture 13: Processor Architecture and Pipelining



The University of
Nottingham

Abstract View of MIPS Implementation

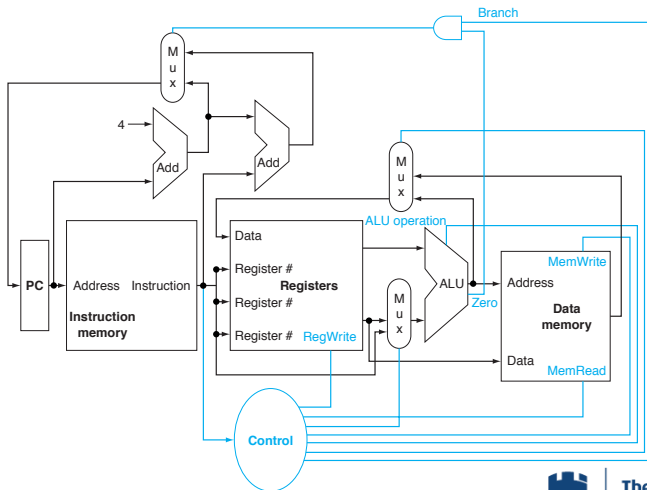


Datapath and Control

- Most instructions have common initial operations
 - Fetch instruction from memory at address PC
 - Decode and select register(s) for subsequent operation
 - Use ALU for: address, arithmetic, logic or comparison
 - Remaining operations differ between instruction classes
- Consider datapaths used in following instruction classes
 - Memory-reference: e.g. lw and sw
 - Arithmetic/logic: e.g. add, sub and slt
 - Branching: e.g. beq and j
- Multiplexors select between multiple data sources
 - Another layer of control logic over previous diagram



Multiplexers and Control Logic



Functional Units and Their Timings

- There are at least five functional units, or stages:
 - IF – Instruction Fetch – get instruction from memory
 - ID – Instruction Decode – get source register operands
 - EX – Execute – ALU operation
 - MEM – Memory Access – data memory read or write
 - WB – Write-Back – result to destination register
- Some stages take longer to finish than others, e.g.

Type	Duration ¹	Stage
Memory	200ps	IF, MEM
ALU	200ps	EX
Register	100ps	ID, WB

¹ $10^{-12}\text{s} = 1\text{ps}$, one picosecond



Critical Paths and Instruction Timings

- Each instruction uses different subset of functional units

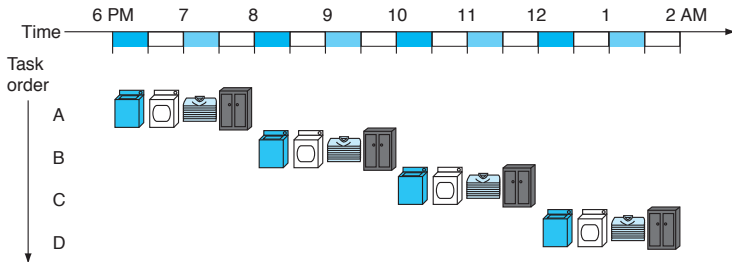
Class	IF	ID	EX	MEM	WB	Total
R-Type	200	100	200		100	600
Load	200	100	200	200	100	800
Store	200	100	200	200		700
Cond. Branch	200	100	200			500
Jump	200					200

- Hence some instructions *could* run faster than others
- But if every instruction must take exactly one cycle,
 - All instructions must take worst-case timing
 - Clock speed will be constrained by slowest instruction



The Laundry Room Analogy

- If it takes 2 hours to wash, dry, fold and store one set of clothes, how long will it take for 20 sets?

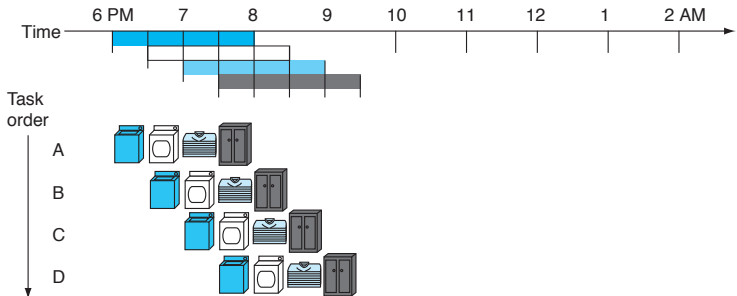


- Total of $2 \times 20 = 40$ hours?



The Laundry Room Analogy

- If it takes 2 hours to wash, dry, fold and store one set of clothes, how long will it take for 20 sets?



- Total of $0.5 \times 20 + 3 \times 0.5 = 11.5$ hours

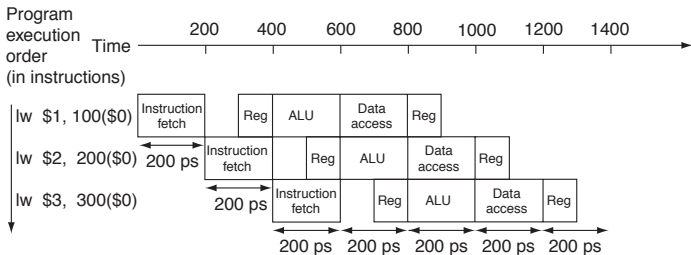


A Production Line for Instructions

- Execute multiple instructions overlapped
 - Make each stage simple and fast; one cycle per stage
 - Start next instruction as soon as current stage is free
 - Same concept as a factory production line
- Instruction *latency* is just as long as before
 - Maybe even a little longer due to pipelining overheads
- But instruction throughput massively increased
 - Throughput is more important than latency
- Ideal case: every instruction with a timing of t can be divided into s stages. Executing n instructions takes,
 - Pipelined, at s/t Hz: $(n + (s - 1))t/s \approx nt/s$
 - Single-cycle, at $1/t$ Hz: nt



Pipeline Overheads



- Even though some stages take less time than others...
- ... speed is still limited by the slowest component
 - Here, slowest stage rather than slowest instruction



Designing ISAs for Pipelining

- Pipelining favours uniform timing and few special cases
- MIPS architecture was designed with pipelining in mind
 - Fixed 32-bit instructions simplifies instruction fetch
 - Few instruction formats, sharing common operand fields
 - Only 1w/sw access memory; ALU calculates address
 - Aligned memory references for single-cycle access
 - Slow instructions like `mult` taken out of pipeline
 - Write to dedicated registers HI and LO (no WB stage)
 - Avoids slowing down the EX stage



Obstacles to Pipelining

- Previously assumed no interaction between instructions
 - Can always issue one instruction every clock cycle
- Reality: various hazards prevents smooth pipeline flow
- Structural Hazards: hardware cannot support instruction
- Data Hazards: ALU needs value not yet in register file
- Control Hazards: IF from PC+4 after branch instruction?



Structural Hazards

- Structural Hazard: hardware cannot support instruction
- Suppose we want to add a new instruction:
 $\text{xor } dst, src_0, n(src_1)$
 - Fetch second operand during MEM, two cycles after EX!
 - Requires an additional MEM (read) stage before EX
 - Requires ALU to calculate $n+src_1$ as well as XOR
 - But each instruction only has one cycle in EX stage!
- Can't simultaneously IF and MEM on same memory bus
 - Switch to a Harvard architecture
 - Dual-ported memory allows two operations each cycle
 - Cache memory often separate for instruction and data
 - Design ISA to avoid structural-hazards!



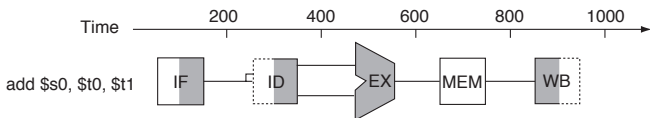
Data Hazards

- Data Hazards: ALU needs value not yet in register file
- Suppose we execute the following dependent instructions:

add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3

- Result of add not written to \$s0 until WB
- But sub requires $\$s0 = \$t0 + \$t1$ the very next cycle!

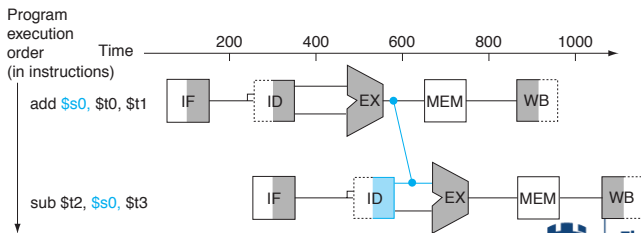


- Stall sub in ID for 3 cycles until result written to \$s0?



EX Forwarding

- Wasted cycles waiting for previous instruction to complete
- Compiler could fill bubbles with independent instructions
 - Or even the hardware – out-of-order execution
 - Hard to find useful instructions; happens too often!
- Better solution – forward result from EX output
 - Extra hardware to take result directly from ALU output



- No stalls required

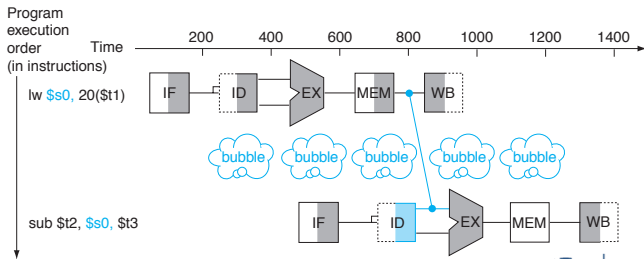


MEM Forwarding

- What about load instructions?
- Consider the following instruction sequence:


```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```

 - Result from `lw` not available until after MEM stage



- Still requires one bubble to be inserted



Reordering Instructions

- Reorder the following to eliminate pipeline stalls:
 lw \$s0, 20(\$t1) lw \$s0, 20(\$t1)
 sub \$t2, \$s0, \$t3 lw \$s1, 24(\$t1)
 sw \$s0, 20(\$t1) sub \$t2, \$s0, \$t3
 lw \$s1, 24(\$t1) sw \$s0, 20(\$t1)
- Now try the example on H&P p378



Branch/Control Hazards

- Control Hazards: IF from PC+4 after branch instruction?
- Consider the following instruction sequence:

```
        beq $s0, $s1, next
                addi $s2, $s2, 1
next:    lw $s0, ($s2)
```
- Which instruction do we fetch after beq?
 - Could stall for 2 cycles until $\$s0 \equiv \$s1$ decided after EX
 - Fetch addi anyway; if wrong, flush/restart the pipeline
- Solutions not as effective as forwarding for data hazards



Branch Prediction

- Static Branch Prediction
 - If target before PC, predict '*taken*' – likely to be a loop
 - Otherwise could be if-then control – predict '*not taken*'
 - Unconditional branches (or 'jumps') always '*taken*'
- Dynamic Branch Prediction
 - Keep track of recent branch decisions
 - If previous branches taken, fetch from branch target
 - Otherwise predict '*not taken*'; fetch from PC+4
- Alternatively, employ *delayed branches*
 - Execute the instruction at PC+4 anyway
 - Instruction following branch called the 'branch delay slot'



Reading Material

- H&P: §5.1 Introduction to *The Processor: Datapath and Control*
- H&P: §6.1, An Overview of Pipelining
- For more detailed information,
 - H&P: §6.5, Data Hazards and Forwarding
 - H&P: §6.6, Branch Hazards

