CS4100: Computer System Design

Fundamentals of Pipelining

Madhu Mutyam
PACE Laboratory
Department of Computer Science and Engineering
Indian Institute of Technology Madras

Sept 14-21, 2015

Pipelining in Admissions Process

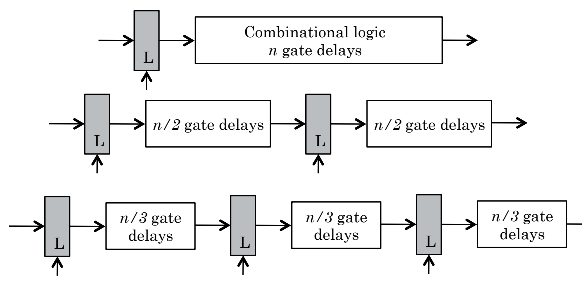
Verification of medical record

Verification of original certificates

Payment of fee

Pipelining

- Pipelining involves partitioning the system into multiple stages with added buffers between the stages
- Performance gained in a pipeline is proportional to the depth of a pipeline
- Pipelining can increase the throughout of a system



$$BW = \frac{1}{n}$$

$$BW = \frac{2}{n}$$

$$BW = \frac{3}{n}$$

- Potential k-fold increase in throughput with k-stage pipeline

Clocking Constraints Limit the Number of Pipeline Stages

- Let F be the combinational part and L be the set of latches
- T_M: The maximum propagation delay through F
- T_m: The minimum propagation delay through F
- T_L: Time needed for proper clocking (*setup*, *hold*, etc)
- T_1 and T_2: The time at which the first and second set of signals applied to the inputs of F

$$T_2 + T_m > T_1 + T_M + T_L$$
$$T_2 - T_1 > T_M - T_m + T_L$$

- (T_M - T_m) and T_L limit the clock rate
- Making the path lengths same brings $T_M - T_m$ close to zero
- The minimum time required for latching and the clock skew limit the depth of the pipeline

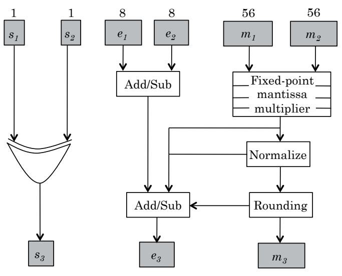
Optimal Pipeline Depth

- Let G and T be the cost and the latency of a non-pipelined design
- The cost of a k-stage pipelined design is $G + kL$, where L is the latch cost
- The latency of a k-stage pipelined design is $\frac{T}{k} + S$, where S is the latch delay
- Let Performance (P) = $\frac{1}{\text{latency}}$

$$\begin{aligned}\frac{Cost_{pipelined\ design}}{Performance_{pipelined\ design}} &= \frac{G+kL}{[\frac{1}{\frac{T}{k}+S}]} \\ &= (G+kL)(\frac{T}{k}+S) \\ &= GS + LT + kLS + \frac{GT}{k} \\ k_{opt} &= \sqrt{\frac{GT}{LS}}\end{aligned}$$

Pipelining in Digital Circuits[a]

[a]Waser and Flynn, 1982

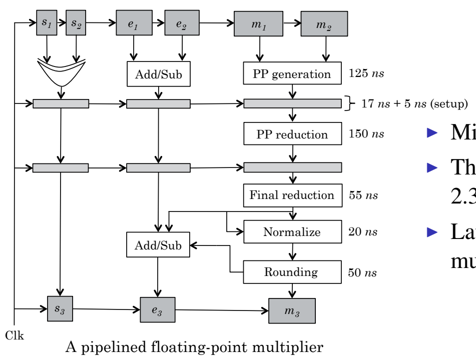


A non-pipelined floating-point multiplier

Module	Delay (*ns*)
Partial product (PP) generation	125
Partial product (PP) reduction	150
Final reduction	55
Normalization	20
Rounding	50
Total	400

Pipelining in Digital Circuits[a]

[a] Waser and Flynn, 1982



A pipelined floating-point multiplier

- Min. clock period is 172 ns
- Throughput increases by 2.3×
- Latency for each multiplication is 516 ns

Pipelining Idealism

Idealism	Realism
Uniform sub-computations	Ex: Pipelined floating-point multiplier Internal fragmentation
Identical computations	Ex: ADD R1, R2, R3 STORE R4, #100 BEQ R1, R5, LABEL External fragmentation
Independent computations	Ex: ADD R1, R2, R3 SUB R4, R1, R5 MUL R6, R1, R4 Pipeline stalls

Instruction Set Architecture Impacts Pipeline Performance

- Uniform sub-computations
  - Memory access is a critical sub-computation
  - Memory addressing modes should be minimised
  - Fast cache memories should be employed
- Identical computations
  - Reduce the complexity and diversity of different instruction types
  - Ex: RISC architecture
- Independent computations
  - Memory addressing modes can make dependency check difficult
  - Registers are explicitly specified in the instruction, and hence register dependencies are easier to check

A Simple Implementation of a RISC Instruction Set

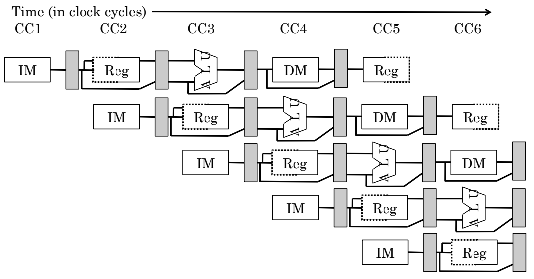
- Instruction fetch cycle (IF)
  - Based on PC value, fetch the instruction from memory
  - Update PC
- Instruction decode/register fetch cycle (ID)
  - Decode the instruction and register read
  - Do the equality check on the registers for a possible branch
  - Compute branch target address, if needed
- Execution/effective address cycle (EX)
  - Memory reference: Calculate the effective address (EA)
  - Register-register ALU instruction
  - Register-immediate ALU instruction

A Simple Implementation of a RISC Instruction Set (Contd...)

- Memory access cycle (MEM)
  - Load instruction: Read data from memory using the EA
  - Store instruction: Write data to memory using the EA
- Write-back cycle (WB)
  - ALU or load instruction: Write result into the register file
- Cycles required to implement different instructions
  - Branch instructions – 2 cycles
  - Store instructions – 4 cycles
  - All other instructions – 5 cycles

5-Stage Pipeline and Datapath for a RISC Processor

Instruction	Clock Cycle No.								
Number	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
$i+1$		IF	ID	EX	MEM	WB			
$i+2$			IF	ID	EX	MEM	WB		
$i+3$				IF	ID	EX	MEM	WB	
$i+4$					IF	ID	EX	MEM	WB



An Example

- Assume that a non-pipelined processor has a 1 *ns* clock cycle and it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, resp. Suppose that due to clock skew and setup, pipelining the processor adds 0.2 *ns* of overhead to the clock. Ignoring any latency impact, how much speedup in instruction execution rate we get from the pipeline design?

$$\begin{aligned}\text{Avg. Execution Time}_{NP} &= \text{Clock Cycle Time} \times \text{Avg. CPI}\\ &= 1\,ns \times [(40\% + 20\%) \times 4 + 40\% \times 5]\\ &= 4.4\,ns\end{aligned}$$

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Avg. Execution Time}_{NP}}{\text{Avg. Execution Time}_{P}}\\ &= \frac{4.4\,ns}{1.2\,ns}\\ &= 3.7\,\times\end{aligned}$$

Pipeline Hazards

- Consequences of the pipeline organization and inter-instruction dependencies
  - Structural hazards – due to resource conflicts
  - Data hazards – due to instruction dependence
  - Control hazards – due to branches and jumps
- Hazards can stall the pipeline

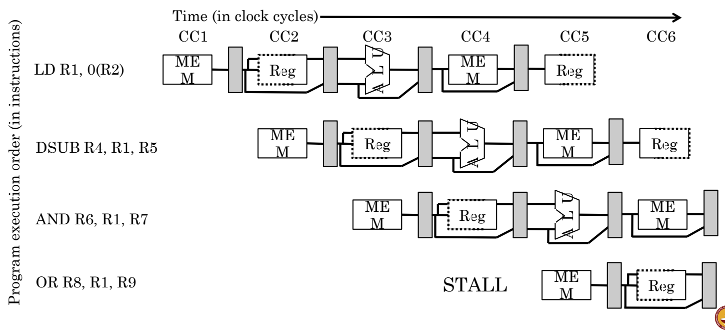
$$\begin{aligned}\text{Speedup}_{\text{pipelining}} &= \frac{CPI_{unpipelined}}{CPI_{Ideal} + \text{Pipeline stall cycles per instruction}}\\ &= \frac{CPI_{unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}\end{aligned}$$

Assuming that all instructions will go through all the pipeline stages,

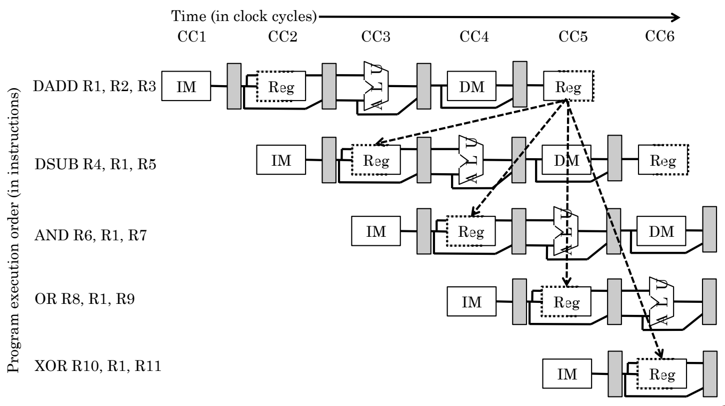
$$\text{Speedup}_{\text{pipelining}} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

Structural Hazards

- Due to non-pipelined functional unit or lack of sufficient number of functional units
- Ex: unified cache with single port:

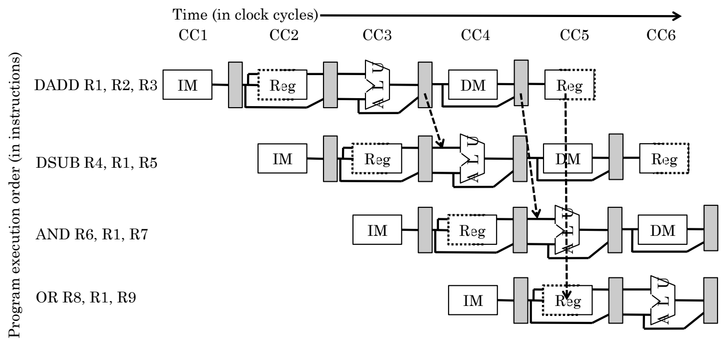


Data Hazards



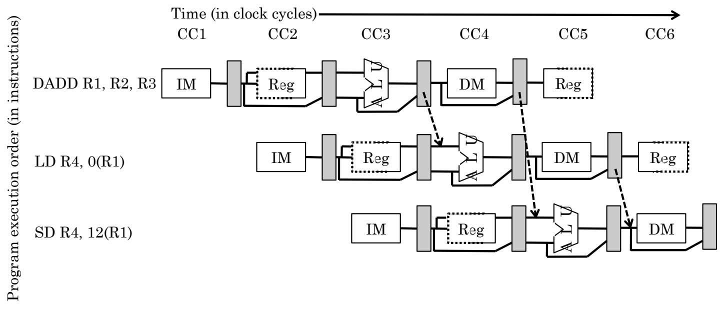
Minimizing Data Hazards

- Operand forwarding

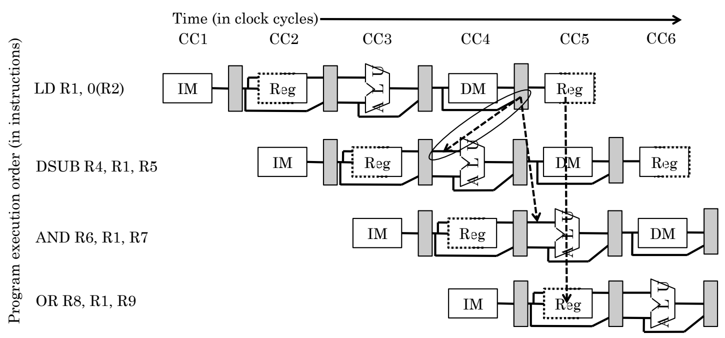


Minimizing Data Hazards

- Operand forwarding for *Stores*



Data Hazards Requiring Stalls



Control Hazards

- Cause higher performance penalty as compared to data hazards
- Branch target address is computed only at the end of ID stage

Branch instr.	IF	ID	EX	MEM	WB		
Branch succ.		**IF**	IF	ID	EX	MEM	WB
Branch succ+1				IF	ID	EX	MEM
Branch succ+2					IF	ID	EX

$$\text{Speedup}_{\text{pipeline}} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Reducing Pipeline Branch Penalties

- Flush the pipeline
  - Delete any instruction after the branch until the branch destination is known
- Predict-not-taken
  - Processor state should not be changed until the branch outcome is definitely known
- Predict-taken
  - Fetch the instruction at target as soon as the branch is decoded and the target address is computed
- Delayed branch
  - The execution cycle with a branch delay of one cycle is:

    Branch inst. $\rightarrow$ sequential succ1 $\rightarrow$ branch target if taken
  - Disallow branch instructions in the branch delay slot

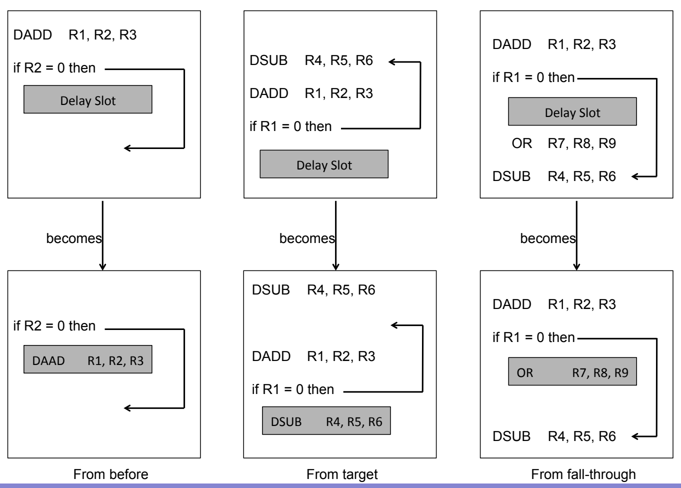
Pipeline Sequence for Predicted-Not-Taken Technique

Not-taken branch instr.	IF	ID	EX	MEM	WB		
Inst. *i*+1		IF	ID	EX	MEM	WB	
Inst. *i*+2			IF	ID	EX	MEM	WB
Inst. *i*+3				IF	ID	EX	MEM
Inst. *i*+4					IF	ID	EX
Taken branch instr.	IF	ID	EX	MEM	WB		
Inst. *i*+1		IF	—	—	—	—	
Branch target			IF	ID	EX	MEM	WB
Branch target+1				IF	ID	EX	MEM
Branch target+2					IF	ID	EX

Pipeline Sequence for Delayed Branch Technique

Not-taken branch instr.	IF	ID	EX	MEM	WB		
Branch delay inst. *i*+1		IF	ID	EX	MEM	WB	
Inst. *i*+2			IF	ID	EX	MEM	WB
Inst. *i*+3				IF	ID	EX	MEM
Inst. *i*+4					IF	ID	EX
Taken branch instr.	IF	ID	EX	MEM	WB		
Branch delay inst. *i*+1		IF	ID	EX	MEM	WB	
Branch target			IF	ID	EX	MEM	WB
Branch target+1				IF	ID	EX	MEM
Branch target+2					IF	ID	EX

Scheduling the Branch Delay Slot



```
DADD   R1, R2, R3
if R2 = 0 then
    Delay Slot
```

becomes

```
if R2 = 0 then
    DAAD   R1, R2, R3
```

```
DSUB   R4, R5, R6
DADD   R1, R2, R3
if R1 = 0 then
    Delay Slot
```

becomes

```
DSUB   R4, R5, R6
DADD   R1, R2, R3
if R1 = 0 then
    DSUB   R4, R5, R6
```

```
DADD   R1, R2, R3
if R1 = 0 then
    Delay Slot
OR     R7, R8, R9
DSUB   R4, R5, R6
```

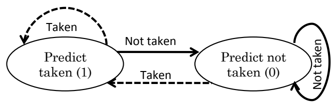
becomes

```
DADD   R1, R2, R3
if R1 = 0 then
    OR     R7, R8, R9
DSUB   R4, R5, R6
```

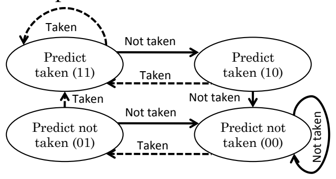
Dynamic Branch Prediction

- Consider a branch outcome sequence (T – *Taken* and N – *Not Taken*):

  T N T N T N T N T N T
- Predict the next outcome of a branch based on its present outcome



- Accuracy is very poor
- 2-bit prediction scheme



- Prediction must miss twice before it is changed
- Branch history table implementation:
  - As a special cache or appending a pair of bits to each cache block

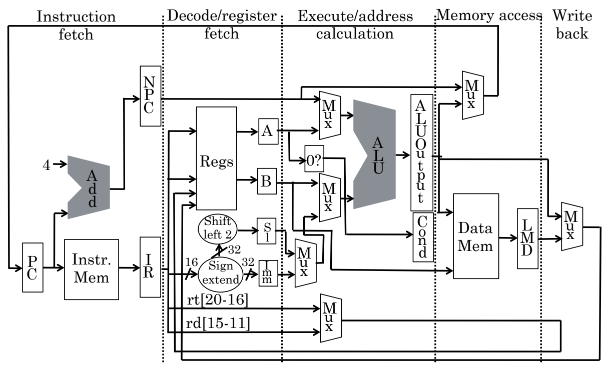
A Simple Implementation of MIPS ISA

- Instruction fetch cycle (IF)
  ```
  IR ← Mem[PC];
  NPC ← PC + 4;
  ```
- Instruction decode/register fetch cycle (ID)
  ```
  A ← Regs[rs];
  B ← Regs[rt];
  Imm ← Sign-extended immediate field of IR;
  ```
- Execution/effective address cycle (EX)
  - Memory reference: `ALUOutput ← A + Imm;`
  - Register-register ALU instruction: `ALUOutput ← A` *Op* `B;`
  - Register-immediate ALU instruction: `ALUOutput ← A` *Op* `Imm;`
  - Branch instruction: `ALUOutput ← NPC + (Imm << 2);`
    `Cond ← (A == 0);`

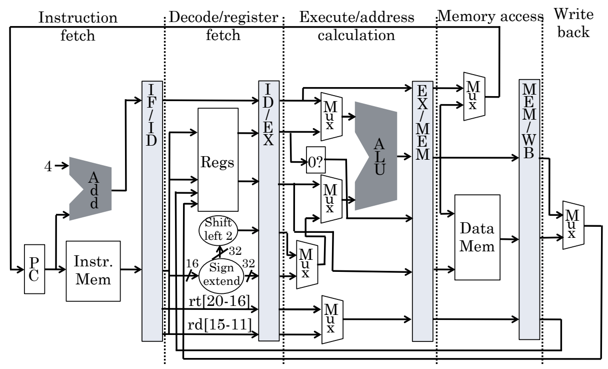
A Simple Implementation of MIPS (Contd)

- Memory access/branch completion cycle (MEM)
  - `PC ← NPC;`
  - Memory reference: `LMD ← Mem[ALUOutput]` or
    `Mem[ALUOutput] ← B;`
  - Branch instruction: `if(Cond) PC ← ALUOutput;`
- Write-back cycle (WB)
  - Register-register ALU instruction: `Regs[rd] ← ALUOutput;`
  - Register-immediate ALU instruction: `Regs[rt] ← ALUOutput;`
  - Load instruction: `Regs[rt] ← LMD;`

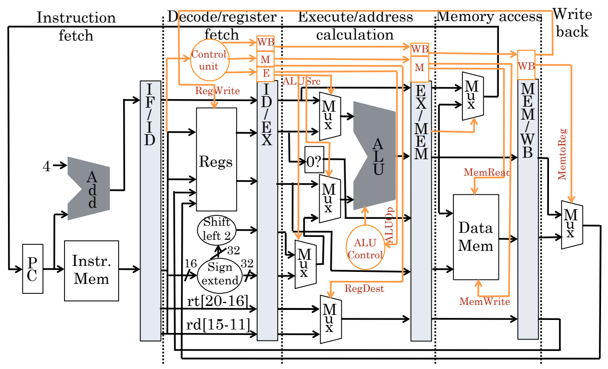
Instruction Flow in the MIPS Data Path



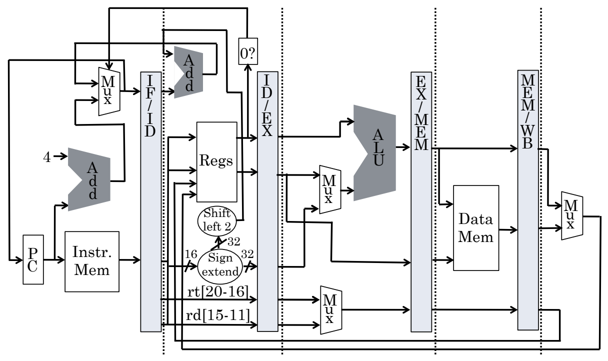
The MIPS Pipelined Data Path



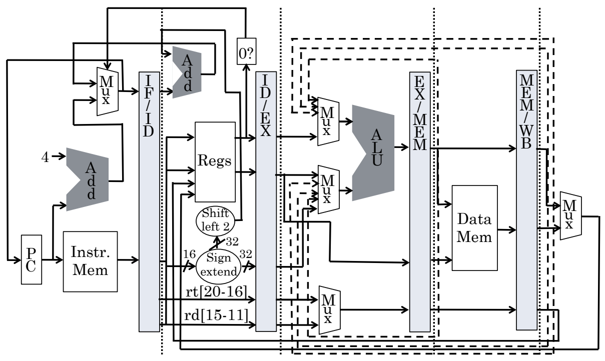
The MIPS Pipelined Data Path + Control Unit



Dealing with Branches in the MIPS Pipeline



Implementing Operand Forwarding in the MIPS Pipeline



Thank You