RISC Pipeline

Han Wang
CS3410, Spring 2010
Computer Science
Cornell University

See: P&H Chapter 4.6
### Homework 2

<table>
<thead>
<tr>
<th>Din[7:0]</th>
<th>DOut [9:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>H G F E D C B A</td>
<td>j h g f i e d c b a</td>
</tr>
</tbody>
</table>

RD (prior) | RD (after)
---|---
+1 | 1 0 0 1 0 0 1 0 1 0 -1
-1 | 1 0 0 1 1 0 1 0 1 0 +1

<table>
<thead>
<tr>
<th>Din[7:0]</th>
<th>DOut [9:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>H G F E D C B A</td>
<td>j h g f i e d c b a</td>
</tr>
</tbody>
</table>

RD (prior) | RD (after)
---|---
+1 | 1 0 0 1 0 0 1 0 1 0 -1
-1 | 1 0 0 1 1 0 1 0 1 0 +1
Announcements

- Homework 2 due tomorrow midnight
- Programming Assignment 1 release tomorrow
- Pipelined MIPS processor (topic of today)
- Subset of MIPS ISA

Feedback

- We want to hear from you!
- Content?
<table>
<thead>
<tr>
<th>op</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
</table>
| 0x3 | JAL      | r31 = PC+8 (+8 due to branch delay slot)  
      |          | PC = (PC+4)$_{31..28}$ | (target $<< 2$)
Review: Single cycle processor
Single Cycle Processor

Advantages

• Single Cycle per instruction make logic and clock simple

Disadvantages

• Since instructions take different time to finish, memory and functional unit are not efficiently utilized.
• Cycle time is the longest delay.
  – Load instruction
• Best possible CPI is 1
Five stage “RISC” load-store architecture

1. Instruction fetch (IF)
   - get instruction from memory, increment PC

2. Instruction Decode (ID)
   - translate opcode into control signals and read registers

3. Execute (EX)
   - perform ALU operation, compute jump/branch targets

4. Memory (MEM)
   - access memory if needed

5. Writeback (WB)
   - update register file

Slides thanks to Sally McKee & Kavita Bala
Break instructions across **multiple clock cycles** (five, in this case)

Design a separate **stage** for the execution performed during each clock cycle

Add **pipeline registers** to isolate signals between different stages
Stage 1: Instruction Fetch

Fetch a new instruction **every cycle**
- Current PC is index to instruction memory
- Increment the PC at end of cycle (assume no branches for now)

Write values of interest to **pipeline register (IF/ID)**
- Instruction bits (for later decoding)
- PC+4 (for later computing branch targets)
IF

INSTRUCTION MEMORY

addr mc

+4

00 = read word

PC

+4

WE

PC+4 inst

Rest of pipeline

IF/ID

PC

pcrel pcabs

pcsel pc

pcreg

Stage 2: Instruction Decode

On every cycle:
- Read IF/ID pipeline register to get instruction bits
- Decode instruction, generate control signals
- Read from register file

Write values of interest to pipeline register (ID/EX)
- Control information, Rd index, immediates, offsets, ...
- Contents of Ra, Rb
- PC+4 (for computing branch targets later)
Stage 3: Execute

On every cycle:
- Read ID/EX pipeline register to get values and control bits
- Perform ALU operation
- Compute targets (PC+4+offset, etc.) in case this is a branch
- Decide if jump/branch should be taken

Write values of interest to pipeline register (EX/MEM)
- Control information, Rd index, ...
- Result of ALU operation
- Value in case this is a memory store instruction

Jump: R1

Branch Predictor

SW R2, R1
MEM
Stage 4: Memory

On every cycle:

- Read EX/MEM pipeline register to get values and control bits
- Perform memory load/store if needed
  - address is ALU result

Write values of interest to pipeline register (MEM/WB)

- Control information, Rd index, ...
- Result of memory operation
- Pass result of ALU operation
MEM
ctrl
MEM/WB

Stage 3: Execute

B
D

EX/MEM

ctrl

D

mem

RAM, mc

addr
d_{in}, d_{out}

Rest of pipeline

MEM/WB

ctrl
Stage 5: Write-back

On every cycle:

- Read MEM/WB pipeline register to get values and control bits
- Select value and write to register file
Stage 4: Memory

MEM/WB

result

dest

MEM/WB

ctrl

M

D

MAX

Adder

dest
Example

```
add r3, r1, r2;
nand r6, r4, r5;
lw r4, 20(r2);
add r5, r2, r5;
sw r7, 12(r3);
```

\[ r3 = r1 + r2 = 45 \]
\[ r6 = \overline{(r4 \& r5)} = \frac{111}{18 \oplus 7} \]
\[ r4 = \text{MEM}[r2 + 20] \]
\[ r5 = \frac{r2 + r5}{9 + 7} = 16 \]
\[ \text{MEM}[r3 + r2] = r7 \]

\[ \ldots 010010 \]
\[ \ldots 0010 \]
\[ \ldots 0001 \]
\[ \ldots 111101 \]
Latency: 5
Throughput: 5
Concurrency: 5

CPI = 1
Powerful technique for masking latencies

• Logically, instructions execute one at a time
• Physically, instructions execute in parallel
  – Instruction level parallelism

Abstraction promotes decoupling

• Interface (ISA) vs. implementation (Pipeline)
The end
Assume eight-register machine

Run the following code on a pipelined datapath

```plaintext
add 3 1 2 ; reg 3 = reg 1 + reg 2
nand 6 4 5 ; reg 6 = \sim\(\text{reg 4 \& reg 5}\)
lw 4 20 (2) ; reg 4 = Mem[reg2+20]
add 5 2 5 ; reg 5 = reg 2 + reg 5
sw 7 12(3) ; Mem[reg3+12] = reg 7
```
Fetch: nand 6 4 5

Time: 2

IF/ID     ID/EX     EX/MEM     MEM/WB

nand 6 4 5    add 3 1 2

PC        Inst mem

MUX

1

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX

MUX
lw 4 20(2)  nand 6 4 5  add 3 1 2

Fetch: lw 4 20(2)

Time: 3
Fetch:
sw 7 12(3)

Time: 5

IF/ID  ID/EX  EX/MEM  MEM/WB

sw 7 12(3)  add 5 2 5  lw 4 20 (2)  nand 6 4 5  add 3 1 2