Designing a Pipelined CPU

Instruction Latencies and Throughput

- Single-Cycle CPU
- Multiple-Cycle CPU
- Pipelined CPU

Pipelining Advantages

- Higher throughput
- Higher of CPU resources

But, more complicated *datapath*, more complex control(?)
Pipelining in Modern CPUs

- CPU Datapath
- Arithmetic Units
- System Buses
- Software (at multiple levels)
- etc...

A Pipelined Datapath

IF: Instruction fetch
ID: Instruction decode and register fetch
EX: Execution and effective address calculation
MEM: Memory access
WB: Write back

Execution in a Pipelined Datapath

IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back
Mixed Instructions in the Pipeline

Pipeline Principles

- All instructions that share a pipeline must have the same __________ in the same __________.
  - therefore, add does nothing during Mem stage
  - sw does nothing during WB stage
- All intermediate values must be latched each cycle.
- There is no functional block reuse

Pipelined Datapath

The Pipeline in Execution

add $10, $1, $2
The Pipeline in Execution

Instruction Fetch  Instruction Decode/ Register Fetch  Execute/ Address Calculation

sub $15, $4, $1
iw $12, 1000($4)

The Pipeline, with controls

Pipelined Control

• can’t use
• not really appropriate.
• !
  – signals generated, but follow instruction through the pipeline

But….
Pipelined Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegSel</th>
<th>ALUOp1</th>
<th>ALUOp2</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>RMemWrite</th>
<th>MemoReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Format</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SW</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Beq</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The Pipeline with Control Logic

Is it really that easy?

- What happens when...
  - add $3, $10, $11
  - lw $8, 1000($3)
  - sub $11, $8, $7
The Pipeline in Execution

```
sub $11, $8, $7  lw $8, 1000($3)  add $3, $10, $11
```

Memory Access  Write Back

```
add $10, $1, $2  sub $11, $8, $7  lw $8, 1000($3)  add $3, $10, $11
```

Instruction  memory  Address

```
4  32
```

Add  Add  result  Shift  left 2

IF/ID  EX/MEM  MEM/WB

```
M  mux
```

Write  data

```
0Write
```

PC

```
0
```

Read  data 1  Read  data 2  Read  register 1  Read  register 2

16  Sign  extend

Write  register  Write  data

```
0Write
```

ALU  result

```
0
```

M  mux  ALU  Zero

ID/EX  Data  memory  Address

```
R2 Available
```

```
CC1  CC2  CC3  CC4  CC5  CC6  CC7  CC8
```

```
sub $2, $1, $3
```

```
and $12, $2, $5
```

```
or $13, $6, $2
```

```
add $14, $2, $2
```

```
sw $15, 100($2)
```

Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.

```
R2 Available
```

```
R2 Needed
```

Pipelining Key Points

- \( ET = IC \times CPI \times CT \)
- We achieve high [_________] without reducing instruction [_________].
- Pipelining exploits a special kind of parallelism (parallelism between functionality required in different cycles).
- Pipelining uses combinational logic to generate (and registers to propagate) control signals.
- Pipelining creates potential hazards.