CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Lecture 30: Pipeline Parallelism 1

Instructor:
Dan Garcia

http://inst.eecs.Berkeley.edu/~cs61c/sp13
Today, computer chips can process data at 10 trillion ($10^{13}$) bits per second. But, even though neurons in the human brain fire at a rate of 100 times per second, the brain still outperforms the best computers at various tasks. The main reason being that calculations done by computer chips happen in isolated pipelines one at a time.
**Datapath Control Signals**

- **ExtOp:** “zero”, “sign”
- **ALUsrc:** 0 ⇒ regB; 1 ⇒ immed
- **ALUctr:** “ADD”, “SUB”, “OR”
- **MemWr:** 1 ⇒ write memory
- **MemtoReg:** 0 ⇒ ALU; 1 ⇒ Mem
- **nPC_sel:** 0 ⇒ “+4”; 1 ⇒ “br”
- **RegDst:** 0 ⇒ “rt”; 1 ⇒ “rd”
- **RegWr:** 1 ⇒ write register
Where Do Control Signals Come From?

Inst Memory
Adr

Instruction<31:0>

Op
Fun

Control

nPC_sel
RegWr
RegDst
ExtOp
ALUSrc
ALUctr
MemWr
MemtoReg

DATA PATH
P&H Figure 4.17
### Summary of the Control Signals (1/2)

<table>
<thead>
<tr>
<th>Inst</th>
<th>Register Transfer</th>
</tr>
</thead>
</table>
| add  | \( R[rd] \leftarrow R[rs] + R[rt] \); PC \( \leftarrow \) PC + 4  
  ALUsrc=RegB, ALUctr=“ADD”, RegDst=rd, RegWr, nPC_sel=“+4” |
| sub  | \( R[rd] \leftarrow R[rs] - R[rt] \); PC \( \leftarrow \) PC + 4  
  ALUsrc=RegB, ALUctr=“SUB”, RegDst=rd, RegWr, nPC_sel=“+4” |
| ori  | \( R[rt] \leftarrow R[rs] + \text{zero\_ext(Imm16)} \); PC \( \leftarrow \) PC + 4  
  ALUsrc=Im, Extop=“Z”, ALUctr=“OR”, RegDst=rt, RegWr, nPC_sel=“+4” |
| lw   | \( R[rt] \leftarrow \text{MEM}[ R[rs] + \text{sign\_ext(Imm16)}] \); PC \( \leftarrow \) PC + 4  
  ALUsrc=Im, Extop=“sn”, ALUctr=“ADD”, MemtoReg, RegDst=rt, RegWr, nPC_sel = “+4” |
| sw   | \( \text{MEM}[ R[rs] + \text{sign\_ext(Imm16)]} \leftarrow R[rs] \); PC \( \leftarrow \) PC + 4  
  ALUsrc=Im, Extop=“sn”, ALUctr = “ADD”, MemWr, nPC_sel = “+4” |
| beq  | if (R[rs] == R[rt]) then PC \( \leftarrow \) PC + sign_ext(Imm16) | 00  
  else PC \( \leftarrow \) PC + 4  
  nPC_sel = “br”, ALUctr = “SUB” |
### Summary of the Control Signals (2/2)

#### Table of Control Signals

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>sub</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RegDst</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>ALUSrc</strong></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td><strong>MemtoReg</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>RegWrite</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>MemWrite</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>nPCsel</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>ExtOp</strong></td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>ALUctr&lt;2:0&gt;</strong></td>
<td>Add</td>
<td>Subtract</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>x</td>
</tr>
</tbody>
</table>

#### Instruction Formats

- **R-type**
  - op (31:26)
  - rs (25:21)
  - rt (20:16)
  - rd (15:11)
  - shamt (10:6)
  - funct (5:0)  
  - add, sub

- **I-type**
  - op (31:26)
  - rs (25:21)
  - rt (20:16)
  - immediate (15:0)  
  - ori, lw, sw, beq

- **J-type**
  - op (31:26)  
  - target address (5:0)  
  - jump

---

See Appendix A for more details.

We Don’t Care :-)
Boolean Exprs for Controller

<table>
<thead>
<tr>
<th>Inst Memory Adr</th>
<th>Instruction&lt;31:0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Op</td>
</tr>
<tr>
<td></td>
<td>Fun</td>
</tr>
<tr>
<td></td>
<td>&lt;26:31&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;0:5&gt;</td>
</tr>
</tbody>
</table>

Op 0-5 are really Instruction bits 26-31
Func 0-5 are really Instruction bits 0-5

- **rtype** = \(~op_5 \cdot ~op_4 \cdot ~op_3 \cdot ~op_2 \cdot ~op_1 \cdot ~op_0\)
- **ori** = \(~op_5 \cdot ~op_4 \cdot op_3 \cdot op_2 \cdot ~op_1 \cdot op_0\)
- **lw** = \(op_5 \cdot ~op_4 \cdot ~op_3 \cdot ~op_2 \cdot op_1 \cdot op_0\)
- **sw** = \(op_5 \cdot ~op_4 \cdot op_3 \cdot ~op_2 \cdot op_1 \cdot op_0\)
- **beq** = \(~op_5 \cdot ~op_4 \cdot ~op_3 \cdot op_2 \cdot ~op_1 \cdot ~op_0\)
- **jump** = \(~op_5 \cdot ~op_4 \cdot ~op_3 \cdot ~op_2 \cdot op_1 \cdot ~op_0\)

- **add** = \(rtype \cdot func_5 \cdot ~func_4 \cdot ~func_3 \cdot ~func_2 \cdot ~func_1 \cdot ~func_0\)
- **sub** = \(rtype \cdot func_5 \cdot ~func_4 \cdot ~func_3 \cdot ~func_2 \cdot func_1 \cdot ~func_0\)

How do we implement this in gates?
Controller Implementation

```
  opcode  func
    |      |
```

“AND” logic

```
  add
  sub
  ori
  lw
  sw
  beq
  jump
```
Boolean Exprs for Controller

RegDst = add + sub
ALUSrc = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nPCsel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq
ALUctr[1] = ori

(assume ALUctr is 00 ADD, 01 SUB, 10 OR)

How do we implement this in gates?
Controller Implementation

```
add
sub
ori
lw
sw
beq
jump

“AND” logic
```

```
opcode
func

“OR” logic
```

```
RegDst
ALUSrc
MemtoReg
RegWrite
MemWrite
nPCsel
Jump
ExtOp
ALUctr[0]
ALUctr[1]
```
Call home, we’ve made HW/SW contact!
Review: Single-cycle Processor

- Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits
Single Cycle Performance

• Assume time for actions are
  – 100ps for register read or write; 200ps for other events

• Clock rate is?

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

• What can we do to improve clock rate?
• Will this improve performance as well?
  Want increased clock rate to mean faster programs
Single Cycle Performance

• Assume time for actions are
  – 100ps for register read or write; 200ps for other events

• Clock rate is?

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

• What can we do to improve clock rate?
• Will this improve performance as well?
  Want increased clock rate to mean faster programs
Gotta Do Laundry

• Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  – Washer takes 30 minutes
  – Dryer takes 30 minutes
  – “Folder” takes 30 minutes
  – “Stasher” takes 30 minutes to put clothes into drawers
Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads
Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!
Pipelining Lessons (1/2)

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload.
- **Multiple** tasks operating simultaneously using different resources.
- Potential speedup = Number pipe stages.
- Time to “fill” pipeline and time to “drain” it reduces speedup: 2.3X v. 4X in this example.
• Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

• Pipeline rate limited by slowest pipeline stage

• Unbalanced lengths of pipe stages reduces speedup

Pipelining Lessons (2/2)
Steps in Executing MIPS

1) **IFtch**: Instruction Fetch, Increment PC
2) **Dcd**: Instruction Decode, Read Registers
3) **Exec**:  
   - Mem-ref: Calculate Address  
   - Arith-log: Perform Operation
4) **Mem**:  
   - Load: Read Data from Memory  
   - Store: Write Data to Memory
5) **WB**: Write Data Back to Register
Single Cycle Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back
• Need registers between stages
  – To hold information produced in previous cycle
More Detailed Pipeline
IF for Load, Store, ...
ID for Load, Store, ...
EX for Load

[Diagram of processor pipeline stages: IF/ID, ID/EX, EX/MEM, MEM/WR, showing data flow and operations such as memory access, register read and write, and arithmetic operations like addition and multiplication.]
MEM for Load
WB for Load – Oops!

Wrong register number
Corrected Datapath for Load
Peer Instruction

1) Thanks to pipelining, I have **reduced the time** it took me to wash my shirt.

2) Longer pipelines are **always a win** (since less work per stage & a faster clock).

3) We can **rely on compilers** to help us avoid data hazards by reordering instrs.

```
<table>
<thead>
<tr>
<th>123</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:  FFF</td>
</tr>
<tr>
<td>b:  FFT</td>
</tr>
<tr>
<td>b:  FTF</td>
</tr>
<tr>
<td>c:  FTT</td>
</tr>
<tr>
<td>c:  TFF</td>
</tr>
<tr>
<td>d:  TFT</td>
</tr>
<tr>
<td>d:  TTF</td>
</tr>
<tr>
<td>e:  TTT</td>
</tr>
</tbody>
</table>
```
So, in conclusion

• You now know how to implement the control logic for the single-cycle CPU.
  – (actually, you already knew it!)

• Pipelining improves performance by increasing instruction throughput: exploits ILP
  – Executes multiple instructions in parallel
  – Each instruction has the same latency

• Next: hazards in pipelining:
  – Structure, data, control