CS 61C:
Great Ideas in Computer Architecture

Lecture 12: Control & Operating Speed
Nick Apology And Comments...

- I wish to apologize for Project 1...
  - I thought it would be cool, but it proved too big
- The amount of **code writing** was appropriate
  - You didn't need to write that much, and it touched on a lot of C code semantics
- The amount of **code understanding** was grossly excessive
  - Although a very useful real-world skill, this was not the intent of the project
- Sorry Not Sorry about the RISC-V programming question on the MT however...
  - I strongly hinted that you needed to understand pointers to function, how structures work, etc....
Agenda

• Review Single-Cycle RISC-V Datapath
• Controller
• Instruction Timing
• Performance Measures
• Introduction to Pipelining
• Pipelined RISC-V Datapath
• And in Conclusion, ...
Recap: Adding branches to datapath
Implementing **JALR** Instruction (I-Format)

- **JALR** rd, rs, immediate
  - Writes PC+4 to Reg[rd] (return address)
  - Sets PC = Reg[rs1] + immediate
  - Uses same immediates as arithmetic and loads
    - *no* multiplication by 2 bytes

### Table

<table>
<thead>
<tr>
<th></th>
<th>imm[11:0]</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>offset[11:0]</td>
<td>base</td>
<td>0</td>
<td>dest</td>
<td>JALR</td>
</tr>
</tbody>
</table>
Datapath with Branches
Adding \texttt{jalr} to datapath
Adding jalr to datapath

Figure showing the datapath with jalr addition.
Implementing `jal` Instruction

- JAL saves PC+4 in Reg[rd] (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within ±$2^{19}$ locations, 2 bytes apart
  - ±$2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost
Adding `jal` to datapath
Adding jal to datapath
Recap: Complete RV32I ISA

RV32I has 47 instructions total. 37 instructions covered in CS61C

Remaining instructions (ex: lui, auipc) can be implemented with no significant additions to the datapath: adding a “pass B” option to the ALU and another immediate decoding option. Rest is all control logic.
Single-Cycle RISC-V RV32I Datapath
Clicker Question

What are proper control signals for the `lui` instruction?

A: BSel=0, ASel=0, WBSel=0
B: BSel=0, ASel=0, WBSel=1
C: BSel=0, ASel=1, WBSel=1
D: BSel=1, ASel=1, WBSel=1
Implementing lui
Announcements...

- Project 3-1 due Sunday, March 10, 23:59
- Project 3-2 Partays scheduled
  - Project 3-2 will be released shortly
  - Friday March 15th 5-7pm in 405 Soda
  - Thursday March 21 7-9pm in 540AB Cory
- 1 on 1 appointments still available
Agenda

• Finish Single-Cycle RISC-V Datapath

• **Controller**

• Instruction Timing

• Performance Measures

• Introduction to Pipelining

• Pipelined RISC-V Datapath

• And in Conclusion, ...
Processor

Control

Datapath
PC
Registers
Arithmetic & Logic Unit (ALU)

Memory
Enable?
Read/Write
Address
Write Data
Read Data

Program
Bytes
Data

Processor-Memory Interface

Lecture 12: Control & Performance
Single-Cycle RISC-V RV32I Datapath
### Control Logic Truth Table (incomplete)

<table>
<thead>
<tr>
<th>Inst[31:0]</th>
<th>BrEq</th>
<th>BrLT</th>
<th>PCsSel</th>
<th>ImmSel</th>
<th>BrUn</th>
<th>ASel</th>
<th>BSel</th>
<th>ALUSel</th>
<th>MemRW</th>
<th>RegWEn</th>
<th>WB Sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>*</td>
<td>*</td>
<td>Reg</td>
<td>Reg</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>sub</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>*</td>
<td>*</td>
<td>Reg</td>
<td>Reg</td>
<td>Sub</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>(R-R Op)</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>*</td>
<td>*</td>
<td>Reg</td>
<td>Reg</td>
<td>(Op)</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>addi</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>I</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>lw</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>I</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>Mem</td>
</tr>
<tr>
<td>sw</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>S</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Write</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>*</td>
<td>+4</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>*</td>
<td>ALU</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>bne</td>
<td>0</td>
<td>*</td>
<td>ALU</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>bne</td>
<td>1</td>
<td>*</td>
<td>+4</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>blt</td>
<td>*</td>
<td>1</td>
<td>ALU</td>
<td>B</td>
<td>0</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>bltu</td>
<td>*</td>
<td>1</td>
<td>ALU</td>
<td>B</td>
<td>1</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>jalr</td>
<td>*</td>
<td>*</td>
<td>ALU</td>
<td>I</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>PC+4</td>
</tr>
<tr>
<td>jal</td>
<td>*</td>
<td>*</td>
<td>ALU</td>
<td>J</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>PC+4</td>
</tr>
<tr>
<td>auipc</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>U</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
</tbody>
</table>
Instruction type encoded using only 9 bits
inst[30], inst[14:12], inst[6:2]

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
</tr>
<tr>
<td>imm[20:10:11]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
</tr>
</tbody>
</table>

Not in CS61C
Control Block Design

11-bit address (inputs)

Inst[30, 14:12, 6:2]  BrEq  BrLT

Combinational Logic Function(s)

PCSel  ImmSel[2:0]  BrUn  ASel  BSel  ALUSel[3:0]  MemRW  RegWEn  WBSel[1:0]

15 data bits (outputs)
Control Realization Options

• ROM
  • “Read-Only Memory”
  • Regular structure
  • Can be easily reprogrammed
    • fix errors
    • add instructions
  • Popular when designing control logic manually

• Combinatorial Logic
  • Today, chip designers use logic synthesis tools to convert truth tables to networks of gates
ROM Controller Implementation

Control Word for `add`
Control Word for `sub`
Control Word for `or`

Controller output (PCSel, ImmSel, …)
Agenda

- Finish Single-Cycle RISC-V Datapath
- Controller
- **Instruction Timing**
- Performance Measures
- Introduction to Pipelining
- Pipelined RISC-V Datapath
- And in Conclusion, ...
Approximate Instruction Timing

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-MEM</td>
<td>Reg Read</td>
<td>ALU</td>
<td>D-MEM</td>
<td>Reg W</td>
<td>800 ps</td>
</tr>
<tr>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>100 ps</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Timing

- Maximum clock frequency
  - \( f_{\text{max}} = \frac{1}{800\text{ps}} = 1.25 \text{ GHz} \)

- Most blocks idle most of the time
  - E.g. \( f_{\text{max,ALU}} = \frac{1}{200\text{ps}} = 5 \text{ GHz}! \)
  - How can we keep ALU busy all the time?
  - 5 billion adds/sec, rather than just 1.25 billion?
  - Idea: Factories use three employee shifts - equipment is always busy!
Agenda

• Finish Single-Cycle RISC-V Datapath
• Controller
• Instruction Timing
• **Performance Measures**
• Introduction to Pipelining
• Pipelined RISC-V Datapath
• And in Conclusion, ...
Performance Measures

• “Our” RISC-V executes instructions at 1.25 GHz
  • 1 instruction every 800 ps

• Can we improve its performance?
  • What do we mean with this statement?
  • Not so obvious:
    • Quicker response time, so one job finishes faster?
    • More jobs per unit time (e.g. web server returning pages)?
    • Longer battery life?
## Transportation Analogy

<table>
<thead>
<tr>
<th></th>
<th>Sports Car</th>
<th>Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passenger Capacity</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>Travel Speed</td>
<td>200 mph</td>
<td>50 mph</td>
</tr>
<tr>
<td>Gas Mileage</td>
<td>5 mpg</td>
<td>2 mpg</td>
</tr>
</tbody>
</table>

### 50 Mile trip:

<table>
<thead>
<tr>
<th></th>
<th>Sports Car</th>
<th>Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Travel Time</td>
<td>15 min</td>
<td>60 min</td>
</tr>
<tr>
<td>Time for 100 passengers</td>
<td>750 min</td>
<td>120 min</td>
</tr>
<tr>
<td>Gallons per passenger</td>
<td>5 gallons</td>
<td>0.5 gallons</td>
</tr>
</tbody>
</table>
# Computer Analogy

<table>
<thead>
<tr>
<th>Transportation</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trip Time</td>
<td>Program execution time ( (\text{latency}) ): e.g. time to update display</td>
</tr>
<tr>
<td>Time for 100 passengers</td>
<td>\textbf{Throughput}: e.g. number of server requests handled per hour</td>
</tr>
<tr>
<td>Gallons per passenger</td>
<td>Energy per task*: e.g. how many movies you can watch per battery charge or energy bill for datacenter</td>
</tr>
</tbody>
</table>

*\textbf{Note}: power is not a good measure, since low-power CPU might run for a long time to complete one task consuming more energy than faster computer running at higher power for a shorter time*
“Iron Law” of Processor Performance

\[
\text{Time} = \text{Instructions} \times \text{Cycles} \times \text{Time}
\]

Program \times Program \times \text{Instruction} \times \text{Cycle}
Instructions per Program

• Determined by
  • Task
  • Algorithm, e.g. $O(N^2)$ vs $O(N)$
  • Programming language
  • Compiler
  • Instruction Set Architecture (ISA)

• Input
  • It is, of course, the halting problem to know how many instructions a program will take in advance
(Average) Clock cycles per Instruction

- Determined by
  - ISA (CISC versus RISC)
  - Processor implementation (or microarchitecture)
    - E.g. for “our” single-cycle RISC-V design, CPI = 1
  - Superscalar processors, CPI < 1 (next lecture)
Time per Cycle (1/Frequency)

• Determined by
  • Processor microarchitecture (determines critical path through logic gates)
  • Technology (e.g. 14nm versus 28nm)
  • Power budget (lower voltages reduce transistor speed)
Speed Tradeoff Example

• For some task (e.g. image compression) …

<table>
<thead>
<tr>
<th></th>
<th>Processor A</th>
<th>Processor B</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>1 Million</td>
<td>1.5 Million</td>
</tr>
<tr>
<td>Average CPI</td>
<td>2.5</td>
<td>1</td>
</tr>
<tr>
<td>Clock rate $f$</td>
<td>2.5 GHz</td>
<td>2 GHz</td>
</tr>
<tr>
<td>Execution time</td>
<td>1 ms</td>
<td>0.75 ms</td>
</tr>
</tbody>
</table>

Processor B is faster for this task, despite executing more instructions and having a lower clock rate!
Energy per Task

\[
\text{Energy} = \text{Instructions} \times \text{Energy}\text{Program} \times \text{Program} \times \alpha \text{Instructions} \times \text{Program} \times C V^2
\]

“Capacitance” depends on technology, microarchitecture, circuit details

Supply voltage, e.g. 1V

Want to reduce capacitance and voltage to reduce energy/task
Energy Tradeoff Example

• “Next-generation” processor (Moore’s law)
  • Capacitance, C: reduced by 15 %
  • Supply voltage, $V_{sup}$: reduced by 15 %
  • Energy consumption: $0.85C \times (0.85V)^2 = 0.63E \Rightarrow -39 \%$

• Significantly improved energy efficiency thanks to
  • Moore’s Law AND
  • Reduced supply voltage
Energy “Iron Law”

- Energy efficiency (e.g., instructions/Joule) is key metric in all computing devices
- For power-constrained systems (e.g., 20MW datacenter), need better energy efficiency to get more performance at same power
- For energy-constrained systems (e.g., 1W phone), need better energy efficiency to prolong battery life

\[
\text{Performance} = \frac{\text{Power}}{\text{Energy Efficiency}} = \frac{\text{Tasks/Second}}{\text{Joules/Second}} \times \text{Tasks/Joule}
\]
End of Scaling

• In recent years, industry has not been able to reduce supply voltage much, as reducing it further would mean increasing “leakage power” where transistor switches don’t fully turn off (more like dimmer switch than on-off switch)
• Also, size of transistors and hence capacitance, not shrinking as much as before between transistor generations
• Power becomes a growing concern – the “power wall”
• Cost-effective air-cooled chip limit around ~150W
Adminstrivia

- XXXX
Agenda

• Finish Single-Cycle RISC-V Datapath
• Controller
• Instruction Timing
• Performance Measures
• Introduction to Pipelining
• Pipelined RISC-V Datapath
• And in Conclusion, ...
Pipelining

- A familiar example:
  - Getting a university degree

- Shortage of Computer scientists (your startup is growing):
  - How long does it take to educate 16,000 students?
Computer Scientist Education

- **Option 1:** *serial*
  - 4000 enter
  - 4000 graduate
  - 4000 graduate
  - 4000 graduate
  - 4000
  - 4 years
  - 4 years
  - 4 years
  - 4 years
  - 16,000 in 16 years, average throughput is 1000/year

- **Option 2:** *pipelining*
  - 4 years
  - 4 years
  - 4 years
  - 4 years
  - 4 years
  - 16,000 in 7 years
  - Steady state throughput is 4000/year
  - Resources used efficiently
  - 4-fold improvement over serial education
Latency versus Throughput

- **Latency**
  - Time from entering college to graduation
  - Serial: 4 years
  - Pipelining: 4 years

- **Throughput**
  - Average number of students graduating each year
  - Serial: 1000
  - Pipelining: 4000

- **Pipelining**
  - Increases throughput (4x in this example)
  - But can *never improve* latency
  - Sometimes worse (additional overhead e.g. for shift transition)
Simultaneous versus Sequential

• What happens *sequentially*?
• What happens *simultaneously*?

4 years

4000 graduate

4000 graduate

4000 graduate

4000 graduate

4000 graduate

4000 graduate

4000 graduate

4000 graduate

4000 graduate

4000 graduate
Agenda

• Finish Single-Cycle RISC-V Datapath
• Controller
• Instruction Timing
• Performance Measures
• Introduction to Pipelining
• **Pipelined RISC-V Datapath**
• And in Conclusion, ...
# Pipelining with RISC-V

<table>
<thead>
<tr>
<th>Phase</th>
<th>Pictogram</th>
<th>$t_{step}$ Serial</th>
<th>$t_{cycle}$ Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td><img src="image" alt="IM" /></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Reg Read</td>
<td><img src="image" alt="Reg" /></td>
<td>100 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>ALU</td>
<td><img src="image" alt="ALU" /></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Memory</td>
<td><img src="image" alt="DM" /></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Register Write</td>
<td><img src="image" alt="Reg" /></td>
<td>100 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>$t_{instruction}$</td>
<td><img src="image" alt="Instruction Pipeline" /></td>
<td>800 ps</td>
<td>1000 ps</td>
</tr>
</tbody>
</table>

Instruction sequence:
- `add t0, t1, t2`
- `or t3, t4, t5`
- `sll t6, t0, t3`
Pipelining with RISC-V

**Single Cycle** | **Pipelining**
---|---
**Timing** | \( t_{\text{step}} = 100 \ldots 200 \text{ ps} \) | \( t_{\text{cycle}} = 200 \text{ ps} \)
Register access only 100 ps | All cycles same length

**Instruction time, \( t_{\text{instruction}} \)** | \( = t_{\text{cycle}} = 800 \text{ ps} \) | 1000 ps

**Clock rate, \( f_s \)** | \( 1/800 \text{ ps} = 1.25 \text{ GHz} \) | \( 1/200 \text{ ps} = 5 \text{ GHz} \)

**Relative speed** | 1 x | 4 x

**Instruction sequence**
- add t0, t1, t2
- or t3, t4, t5
- sll t6, t0, t3
Weaver
Sequential vs Simultaneous

What happens sequentially, what happens simultaneously?

$\text{t}_{\text{instruction}} = 1000 \text{ ps}$

add t0, t1, t2
or t3, t4, t5
sll t6, t0, t3
sw t0, 4(t3)
lw t0, 8(t3)
addi t2, t2, 1

$t_{\text{cycle}} = 200 \text{ ps}$
Agenda

- Finish Single-Cycle RISC-V Datapath
- Controller
- Instruction Timing
- Performance Measures
- Introduction to Pipelining
- Pipelined RISC-V Datapath
- And in Conclusion, ...
And in Conclusion, ...

- **Controller**
  - Tells universal datapath how to execute each instruction

- **Instruction timing**
  - Set by instruction complexity, architecture, technology
  - Pipelining increases clock frequency, “instructions per second”
    - But does not reduce time to complete instruction

- **Performance measures**
  - Different measures depending on objective
    - Response time
    - Jobs / second
    - Energy per task