Reminder: Homeworks for Next Two Weeks

- Homework 1 solutions
  - Posted online

- Homework 2
  - Already out
  - Due February 13
  - ISA concepts, ISA vs. microarchitecture, microcoded machines
Reminder: Lab Assignments 1 and 2

- Lab Assignment 1
  - Due this Friday (Feb 3), at the end of Friday lab

- Lab Assignment 1.5
  - Verilog practice
  - Not to be turned in

- Lab Assignment 2
  - Will be out today
  - Due Friday, Feb 17, at the end of the lab
Readings for Next Lecture

- P&P, Revised Appendix C
  - Microarchitecture of the LC-3b
  - Appendix A (LC-3b ISA) will be useful in following this

- P&H, Appendix D
  - Mapping Control to Hardware

- Optional
Answers to Some of Your Questions

- Can you do the labs in System Verilog?
  - Unfortunately, no, you cannot do the Labs in System Verilog.
  - Fortunately, Verilog is easy after you know System Verilog.

- Yes, this class will hopefully fulfill your desire to understand how a processor truly works.

- Yes, this course will be time consuming 😊
  - But, it will also be fun and “easy” (that is if you understand all).

- Yes, I do have your pictures already, but ...

- Yes, you got the MIPS acronym, correct. And, thanks!
Review of Last Lecture: ISA Tradeoffs

- Complex vs. simple instructions: concept of semantic gap
- Use of *translation* to change the tradeoffs
- Fixed vs. variable length, uniform vs. non-uniform decode
- Number of registers, addressing modes
- Unaligned memory access

What is the benefit of translating complex instructions to “simple instructions” before executing them?
- In hardware (a la Intel, AMD)?
- In software (a la Transmeta)?

Which ISA is easier to extend: fixed length or variable length?
How can you have a variable length, uniform decode ISA?
A Note on Length and Uniformity

- Uniform decode usually goes with fixed length

- In a variable length ISA, uniform decode can be a property of instructions of the same length
  - It is hard to think of it as a property of instructions of different lengths
A Note on RISC vs. CISC

- Usually, ...

- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes

- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes
A Note on Small Semantic Gap

- With a small semantic gap
  - Software’s job to translate from HLL to ISA is easy if the HLL constructs map to ISA instructions easily

- Software’s job is harder is the HLL constructs do not map to ISA constructs easily
  - Semantic mismatch

- The semantic gap is small if the semantic level of HLL and ISA are close and the semantics match
Implementing the ISA: Microarchitecture Basics
How Does a Machine Process Instructions?

- What does processing an instruction mean?
- Remember the von Neumann model

$\text{AS} = \text{Architectural (programmer visible) state before an instruction is processed}$

Process instruction

$\text{AS'} = \text{Architectural (programmer visible) state after an instruction is processed}$

- Processing an instruction: Transforming A to A’ according to the ISA specification of the instruction
The “Process instruction” Step

ISA specifies abstractly what A’ should be, given an instruction and A

- It defines an abstract finite state machine where
  - State = programmer-visible state
  - Next-state logic = instruction execution specification

- From ISA point of view, there are no “intermediate states” between A and A’ during instruction execution
  - One state transition per instruction

Microarchitecture implements how A is transformed to A’

- There are many choices in implementation
- We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
  - Choice 1: $AS \rightarrow AS'$ (transform A to A’ in a single clock cycle)
  - Choice 2: $AS \rightarrow AS+MS1 \rightarrow AS+MS2 \rightarrow AS+MS3 \rightarrow AS'$ (take multiple clock cycles to transform AS to AS’)

A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - No intermediate, programmer-invisible state updates

\[ \text{AS} = \text{Architectural (programmer visible) state at the beginning of a clock cycle} \]

Process instruction in one clock cycle

\[ \text{AS'} = \text{Architectural (programmer visible) state at the end of a clock cycle} \]
A Very Basic Instruction Processing Engine

- Single-cycle machine

What is the *clock cycle time* determined by?

What is the *critical path* of the combinational logic determined by?
Remember: Programmer Visible (Architectural) State

Memory
array of storage locations indexed by an address

\[ M[0] \]
\[ M[1] \]
\[ M[2] \]
\[ M[3] \]
\[ M[4] \]
\[ M[N-1] \]

Program Counter
memory address of the current instruction

Registers
- given special names in the ISA (as opposed to addresses)
- general vs. special purpose

Instructions (and programs) specify how to transform the values of programmer visible state
Single-cycle vs. Multi-cycle Machines

- **Single-cycle machines**
  - Each instruction takes a single clock cycle
  - All state updates made at the end of an instruction’s execution
  - **Big disadvantage:** The slowest instruction determines cycle time → long clock cycle time

- **Multi-cycle machines**
  - Instruction processing broken into multiple cycles/stages
  - State updates can be made during an instruction’s execution
  - Architectural state updates made only at the end of an instruction’s execution
  - **Advantage over single-cycle:** The slowest “stage” determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level
Instruction Processing “Cycle”

- Instructions are processed under the direction of a “control unit” step by step.
- Instruction cycle: Sequence of steps to process an instruction
- Fundamentally, there are six phases:
  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result

- Not all instructions require all six stages (see P&P Ch. 4)
Instruction Processing “Cycle” vs. Machine Clock Cycle

- Single-cycle machine:
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete

- Multi-cycle machine:
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, *each phase can take multiple clock cycles to complete*
Instruction Processing Viewed Another Way

- Instructions transform Data (AS) to Data’ (AS’)
- This transformation is done by functional units
  - Units that “operate” on data
- These units need to be told what to do to the data

- An instruction processing engine consists of two components
  - Datapath: Consists of hardware elements that deal with and transform data signals
    - functional units that operate on data
    - hardware structures (e.g. wires and muxes) that enable the flow of data into the functional units and registers
    - storage units that store data (e.g., registers)
  - Control logic: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data
Single-cycle vs. Multi-cycle: Control & Data

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as data signals are operated on
  - Everything related to an instruction happens in one clock cycle

- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the previous cycle
  - Latency of control processing can be overlapped with latency of datapath operation

- We will see the difference clearly in microprogrammed multi-cycle microarchitecture
Many Ways of Datapath and Control Design

- There are many ways of designing the data path and control logic
  
- Single-cycle, multi-cycle, pipelined datapath and control
  
- Single-bus vs. multi-bus datapaths
  - See your homework 2 question
  
- Hardwired/combinational vs. microcoded/microprogrammed control
  - Control signals generated by combinational logic versus
  - Control signals stored in a memory structure

- Control signals and structure depend on the datapath design
A Single-Cycle Microarchitecture
A Closer Look
Remember...

- Single-cycle machine

[Diagram: A circle labeled "Combinational Logic" is connected to a box labeled "Sequential Logic (State)"]
Let’s Start with the State Elements

- **Data and control inputs**

```plaintext
<table>
<thead>
<tr>
<th>Instruction memory</th>
<th>Instruction address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
For Now, We Will Assume

- “Magic” memory and register file
- Combinational read
  - output of the read data port is a combinational function of the register file contents and the corresponding read select port
- Synchronous write
  - the selected register is updated on the positive edge clock transition when write enable is asserted
    - Cannot affect read output in between clock edges
    - Can affect read output at clock edges (but who cares?)
- Single-cycle, synchronous memory
  - Contrast this with memory that tells when the data is ready
  - i.e., Ready bit: indicating the read or write is done
Instruction Processing

- 5 generic steps (P&H)
  - Instruction fetch (IF)
  - Instruction decode and register operand fetch (ID/RF)
  - Execute/Evaluate memory address (EX/AG)
  - Memory operand fetch (MEM)
  - Store/writeback result (WB)

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
**What Is To Come: The Full Datapath**

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
Single-Cycle Datapath for
Arithmetic and Logical Instructions
R-Type ALU Instructions

- Assembly (e.g., register-register signed addition)
  \[ \text{ADD } \text{rd}_{\text{reg}} \text{ rs}_{\text{reg}} \text{ rt}_{\text{reg}} \]

- Machine encoding

<table>
<thead>
<tr>
<th>R-type</th>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>6-bit</td>
</tr>
</tbody>
</table>

- Semantics

if \( \text{MEM[PC]} == \text{ADD} \text{ rd } \text{ rs } \text{ rt} \)

\( \text{GPR[rd]} \leftarrow \text{GPR[rs]} + \text{GPR[rt]} \)

\( \text{PC} \leftarrow \text{PC} + 4 \)
**ALU Datapath**

if MEM[PC] == ADD rd rs rt
GPR[rd] ← GPR[rs] + GPR[rt]
PC ← PC + 4

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**

Combinational state update logic
I-Type ALU Instructions

- Assembly (e.g., register-immediate signed additions)
  \[ \text{ADDI } rt_{reg} \text{ rs}_{reg} \text{ immediate}_{16} \]

- Machine encoding

  \[
  \begin{array}{cccc}
  \text{ADDI} & \text{rs} & \text{rt} & \text{immediate} \\
  \text{6-bit} & \text{5-bit} & \text{5-bit} & \text{16-bit}
  \end{array}
  \]

- Semantics

  if \( \text{MEM[PC]} = \text{ADDI } rt \text{ rs} \text{ immediate} \)
  \[
  \begin{align*}
  \text{GPR}[rt] & \leftarrow \text{GPR}[rs] + \text{sign-extend (immediate)} \\
  \text{PC} & \leftarrow \text{PC} + 4
  \end{align*}
  \]
if MEM[PC] == ADDI rt rs immediate
GPR[rt] ← GPR[rs] + sign-extend (immediate)
PC ← PC + 4

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Single-Cycle Datapath for Data Movement Instructions
Load Instructions

- **Assembly** (e.g., load 4-byte word)
  
  \[
  \text{LW } rt_{\text{reg}} \text{ offset}_{16} (\text{base}_{\text{reg}}) 
  \]

- **Machine encoding**

<table>
<thead>
<tr>
<th>LW</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

- **Semantics**

  if \( \text{MEM}[\text{PC}] = \text{LW } rt \text{ offset}_{16} (\text{base}) \)

  \[
  \text{EA} = \text{sign-extend}(\text{offset}) + \text{GPR}[\text{base}] 
  \]

  \[
  \text{GPR}[rt] \leftarrow \text{MEM}[ \text{translate}(\text{EA}) ] 
  \]

  \[
  \text{PC} \leftarrow \text{PC} + 4 
  \]
if MEM[PC]==LW rt offset_{16} (base) 
EA = sign-extend(offset) + GPR[base] 
GPR[rt] ← MEM[ translate(EA) ] 
PC ← PC + 4
Store Instructions

- Assembly (e.g., store 4-byte word)
  \[ \text{SW } r_{reg} \text{ offset}_{16} (\text{base}_{reg}) \]

- Machine encoding

<table>
<thead>
<tr>
<th>SW</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

  I-type

- Semantics

  if \( \text{MEM}[\text{PC}] = \text{SW } r_{t} \text{ offset}_{16} (\text{base}) \)
  
  \[ \text{EA} = \text{sign-extend}(\text{offset}) + \text{GPR}[\text{base}] \]
  
  \[ \text{MEM}[ \text{translate}(\text{EA}) ] \leftarrow \text{GPR}[rt] \]
  
  \[ \text{PC} \leftarrow \text{PC} + 4 \]
if MEM[PC] == SW rt offset $16_{16}$ (base)

$EA = \text{sign-extend}(\text{offset}) + \text{GPR}[\text{base}]$

$\text{MEM[translate}(EA) \leftarrow \text{GPR}[rt]$

$\text{PC} \leftarrow \text{PC} + 4$

Combination state update logic

[Diagram of SW Datapath with logical blocks and flow of data]
Load-Store Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Datapath for Non-Control-Flow Insts.

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Single-Cycle Datapath for

Control Flow Instructions
Unconditional Jump Instructions

- **Assembly**
  \[ J \text{ immediate}_{26} \]

- **Machine encoding**

  \[
  \begin{array}{c|c}
  J & \text{immediate} \\
  \hline
  6\text{-bit} & 26\text{-bit}
  \end{array}
  \]

  J-type

- **Semantics**

  if \( MEM[PC] == J \text{ immediate}_{26} \)

  \[
  \text{target} = \{ \text{PC}[31:28], \text{immediate}_{26}, 2'b00 \} \\
  \text{PC} \leftarrow \text{target}
  \]
Unconditional Jump Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

if MEM[PC]==J immediate26
PC = { PC[31:28], immediate26, 2’ b00 }
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Conditional Branch Instructions

- Assembly (e.g., branch if equal)
  \[ \text{BEQ} \ rs_{\text{reg}} \ rt_{\text{reg}} \text{ immediate}_{16} \]

- Machine encoding

<table>
<thead>
<tr>
<th>BEQ</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

- Semantics (assuming no branch delay slot)

  if \( \text{MEM}[\text{PC}] == \text{BEQ} \ rs \ rt \text{ immediate}_{16} \)
  \[
  \text{target} = \text{PC} + 4 + \text{sign-extend(\text{immediate})} \times 4
  \]
  if \( \text{GPR}[\text{rs}] == \text{GPR}[\text{rt}] \) then \( \text{PC} \leftarrow \text{target} \)
  else \( \text{PC} \leftarrow \text{PC} + 4 \)
Conditional Branch Datapath (For You to Fix)

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

**How to uphold the delayed branch semantics?**
Putting It All Together

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
Single-Cycle Control Logic
Single-Cycle Hardwired Control

- As combinational function of Inst=MEM[PC]

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td>6-bit</td>
</tr>
</tbody>
</table>

  R-type

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
</tbody>
</table>

  I-type

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>immediate</td>
</tr>
</tbody>
</table>

  J-type

- Consider
  - All R-type and I-type ALU instructions
  - LW and SW
  - BEQ, BNE, BLEZ, BGTZ
  - J, JR, JAL, JALR
## Single-Bit Control Signals

<table>
<thead>
<tr>
<th></th>
<th>When De-asserted</th>
<th>When asserted</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RegDest</strong></td>
<td>GPR write select according to ( rt ), i.e.,\n</td>
<td></td>
<td>( \text{inst}[20:16] )</td>
</tr>
<tr>
<td><strong>ALUSrc</strong></td>
<td>( 2^{\text{nd}} ) ALU input from ( 2^{\text{nd}} )\n</td>
<td></td>
<td>( \text{GPR read port} )</td>
</tr>
<tr>
<td><strong>MemtoReg</strong></td>
<td>Steer ( \text{ALU result to GPR write port} )\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RegWrite</strong></td>
<td>GPR write disabled</td>
<td>GPR write enabled</td>
<td>( \text{(opcode)!=} \text{SW} &amp;&amp; \text{(opcode)!=} \text{Bxx} &amp;&amp; \text{(opcode)!=} \text{J} &amp;&amp; \text{(opcode)!=} \text{JR} )</td>
</tr>
</tbody>
</table>

JAL and JALR require additional \( \text{RegDest} \) and \( \text{MemtoReg} \) options
## Single-Bit Control Signals

<table>
<thead>
<tr>
<th>PCSrc&lt;sub&gt;1&lt;/sub&gt;</th>
<th>When De-asserted</th>
<th>When asserted</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemRead</td>
<td>Memory read disabled</td>
<td>Memory read port return load value</td>
<td>opcode==LW</td>
</tr>
<tr>
<td>MemWrite</td>
<td>Memory write disabled</td>
<td>Memory write enabled</td>
<td>opcode==SW</td>
</tr>
<tr>
<td>PCSrc&lt;sub&gt;2&lt;/sub&gt;</td>
<td>According to PCSrc&lt;sub&gt;2&lt;/sub&gt;</td>
<td>next PC is based on 26-bit immediate jump target</td>
<td>(opcode==J)</td>
</tr>
<tr>
<td></td>
<td>next PC = PC + 4</td>
<td>next PC is based on 16-bit immediate branch target</td>
<td>(opcode==Bxx) &amp;&amp; “bcond is satisfied”</td>
</tr>
</tbody>
</table>

JR and JALR require additional PCSrc options
ALU Control

- case opcode

  ‘0’ ⇒ select operation according to funct
  ‘ALUi’ ⇒ selection operation according to opcode
  ‘LW’ ⇒ select addition
  ‘SW’ ⇒ select addition
  ‘Bxx’ ⇒ select bcond generation function
  __ ⇒ don’t care

- Example ALU operations
  - ADD, SUB, AND, OR, XOR, NOR, etc.
  - bcond on equal, not equal, LE zero, GT zero, etc.
R-Type ALU

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
I-Type ALU

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Branch Not Taken

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Branch Taken

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**