CSE 220: System Fundamentals I

Unit 12: MIPS Architecture: Single-cycle Processors

Instructor: Prof. Kevin McDonnell
Microarchitecture

• For most of the remainder of the semester we will look at different ways to piece together a MIPS CPU
• We will study **microarchitecture**, which is how an architecture is actually implemented in the hardware
  • A particular architecture could be implemented in a variety of ways that trade off performance, cost and complexity
• More specifically, we need to determine for a CPU:
  • **datapath**: the functional blocks that do the “work” of the CPU (memory, registers, ALU, muxes)
  • **control**: the control signals that direct the functional blocks to perform the desired operations
Microarchitecture

• We will explore two implementations for MIPS architecture:
  • **Single-cycle**: each instruction executes in a single cycle
  • **Pipelined**: each instruction broken up into series of steps, and multiple instructions execute simultaneously
• Any microarchitecture must represent the **architectural state** of the CPU, which consists of (the values of) the program counter, the 32 registers and the memory
• The control unit receives the current instruction from the datapath and tells the datapath how to execute the instruction
Microarchitecture

• We'll start with the design of the single-cycle processor

• In the beginning we will consider only a subset of MIPS instructions:
  • R-type instructions: \texttt{and, or, add, sub, slt}
  • Memory instructions: \texttt{lw, sw}
  • Branch instructions: \texttt{beq}

• Later we will add other MIPS instructions to the datapath

• We will even invent our own instructions, add them to the architecture and see how the control and datapath must be modified to handle them
State Elements

- These are the units that store values and represent the architectural state.
- In these diagrams, a black line indicates data buses, whereas blue lines indicate control signals.
- Although the instructions and data are stored in different “memories”, in a real computer these would be two different CPU caches, not main memories.
State Elements

- PC gives the address of the current instruction, and PC’ gives the address of the next instruction.
- CLK is the system clock, which oscillates between low and high voltage to synchronize the elements.
- State elements change their state only on the rising edge of the clock (transition from low to high), so all elements change state simultaneously.
State Elements

- The instruction memory has a single read port (RD). The 32-bit instruction address input (A) determines which instruction is sent out on RD.
- The register file contains thirty-two 32-bit registers, two read ports and one write port. A1, A2 and A3 take 5-bit addresses to indicate which registers should be read (A1, A2 for RD1, RD2) and/or written (A3 for WD3)
State Elements

- The *WE3* is the **write enable** control signal. When 1, the register file writes the data into the specified register (given by *A3*) on the rising edge of the clock.
- Note that a register is always outputting a value, so there is no delay in reading the contents of a register.
  - Writing to (updating) a register, though, *does* induce a delay. More on this timing issue later.
State Elements

- Finally, the data memory has a single read/write port. If $WE$ is asserted, the data from $WD$ is written into address $A$ on the rising edge. Otherwise, $WE$ is 0 and the memory reads the data at address $A$ and sends it out on $RD$. 
Single-Cycle Datapath

- Our first iteration of a MIPS CPU will feature a single-cycle datapath, which means that the CPU executes only one instruction per cycle.
- The first step is to read the instruction from instruction memory.

- The instruction memory reads out (fetches) the 32-bit instruction, labeled $Instr$. 

![Diagram of single-cycle datapath](image-url)
Single-Cycle Datapath

• Let’s consider the \texttt{lw} instruction
• \texttt{lw} is an I-type instruction
• \texttt{lw \ rt, imm(rs)}

\textbf{I-type}

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

• \(\text{Reg}[rt] = \text{Mem}[\text{Reg}[rs]+\text{SignImm}]\)

• You will need to know the instruction formats for the final exam, so start studying them now
Single-Cycle Datapath: \texttt{lw}

- For \texttt{lw}, we need to read the source register containing the base address
- This is \texttt{rs}, i.e., $Instr_{25:21}$. We need to send these bits to the register read port $A1$, which reads the register value onto $RD1$. 

![Datapath Diagram](image_url)
Single-Cycle Datapath: \( lw \)

- \( lw \) also needs an offset, which is given in \( Instr_{15:0} \)
- This (signed) offset must be sign-extended. We call this result \( SignImm \).
Single-Cycle Datapath: \( \text{lw} \)

- Now, this offset must be added to the base address that came out of \( RD1 \), so we need an adder.
- \( ALUControl \) is a 3-bit control signal that tells the ALU what operation to perform. 010 here indicates addition.
Single-Cycle Datapath: \( \text{l} w \)

- The output of the ALU gives the effective address of the word we want to read from memory, so this address is sent to the data memory (via \( A \)), which reads the word from memory and writes it to the ReadData bus.
Single-Cycle Datapath: $lw$

- The $ReadData$ bus carries the data word to the register file’s write port, $WD3$
- The $rt$ field of the instruction ($Instr_{20:16}$) is sent to $A3$, which indicates the register that will receive the data
Single-Cycle Datapath: \( \text{lw} \)

- The control signal \( \text{RegWrite} \) is asserted so that the data value is written into the register file on the rising edge of the clock at the end of the cycle.
Single-Cycle Datapath: \( \downarrow w \)

- While all of this is happening, the CPU must compute the address of the next instruction, PC′
- We use an adder to add 4 to the PC and write the new value on the rising edge of the clock
Single-Cycle Datapath: \texttt{sw}

- \texttt{sw} is executed in a similar way to \texttt{lw} and uses much of the same datapath:
  \[
  \texttt{sw} \ rt, \ \text{imm}(rs)
  \]
- \[
  \text{Mem}[\text{Reg}[rs]+\text{SignImm}] = \text{Reg}[rt]
  \]
Single-Cycle Datapath: $SW$

- The *RegWrite* signal is not asserted because we are reading from the register file, not writing to it. However, data is still read from memory, but *ReadData* is simply ignored.

- The *MemWrite* signal is asserted because we are writing to the memory.
Single-Cycle Datapath: R-Type

• Now we'll see how to extend the datapath to handle the R-type instructions add, sub, and, or and slt.
• All of these read two registers, perform an ALU operation, and write back to the register file.
• Example: $\text{Reg}[rd] = \text{Reg}[rs] + \text{Reg}[rt]$ for add.
• So, all will use the same hardware but have different ALUControl signals.

<table>
<thead>
<tr>
<th>R-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I-type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: R-Type

• First, a mux has been added to let us control the source of the $SrcB$ value to the ALU (control signal $ALUSrc$)
• For an R-type instruction the operand comes from the register file
Single-Cycle Datapath: R-Type

• For the \texttt{lw} and \texttt{sw} instructions the operand comes from the instruction itself
• So \textit{ALUSrc} is 0 for R-type instructions and 1 for \texttt{lw} and \texttt{sw}
• We will use muxes again for selecting the source of inputs
Single-Cycle Datapath: R-Type

- When doing a `lw`, we needed to write to the register file
- R-type instructions also write to the register file
- We add a mux and control signal `MemtoReg` to choose between `ReadData` and `ALUResult`
Single-Cycle Datapath: R-Type

- *MemtoReg* is 0 for R-type instructions to choose *Result* from the *ALUResult*.
- *MemtoReg* is 1 for *lw* to choose *ReadData*. We don’t care about the value of *MemtoReg* for *sw* because *sw* does not write to the register file.
Single-Cycle Datapath: R-Type

- For an R-type instruction we write to register $\text{rd} (Instr_{15:11})$

  **R-type**

  \[
  \begin{array}{cccccc}
  \text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
  6 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 6 \text{ bits}
  \end{array}
  \]

- For \text{lw}, we write data to register $\text{rt} (Instr_{20:16})$

  **I-type**

  \[
  \begin{array}{cccc}
  \text{op} & \text{rs} & \text{rt} & \text{imm} \\
  6 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 16 \text{ bits}
  \end{array}
  \]
Single-Cycle Datapath: R-Type

- We add a control signal \( \text{RegDst} \) to choose \( \text{WriteReg} \) from the appropriate field of the instruction.
- \( \text{RegDst} \) is 1 for R-type instructions (to select \( rd, \text{Instr}_{15:11} \)) and 0 for \( \text{lw} \) (to select \( rt, \text{Instr}_{20:16} \)).
- We don’t care about the value of \( \text{RegDst} \) for \( \text{sw} \).
Single-Cycle Datapath: \texttt{beq}

- \texttt{beq} is an I-type instruction
- \texttt{beq} compares two registers and adds a branch offset (with the help of $Instr_{15:0}$) to the PC if the registers’ contents are equal
- To test for equality, we compute $(Reg[rs] - Reg[rt])$ and check if the difference ($ALUResult$) equals zero
- So $ALUControl$ is 110 to direct the ALU to do subtraction
- $Instr_{15:0}$ is the number of instructions to skip over, so we sign-extend the offset and multiply it by 4: $PC' = PCBranch = PC + 4 + \text{SignImm} \times 4$
Single-Cycle Datapath: \texttt{beq}

- Normally, $PC' = PC + 4$, so we need to add a mux to choose which formula to use.
- If control signal \textit{Branch} is asserted and the ALU produces 0, then $PC_{Src}$ is 1 and we choose $PC_{Branch}$. 
Single-Cycle Datapath: \texttt{beq}

- \texttt{ALUSrc} is 0 to select \texttt{SrcB} from the register file
- \texttt{RegWrite} and \texttt{MemWrite} are both 0 (no writing takes place)
- We don’t care about \texttt{RegDst} and \texttt{MemtoReg}'s values because we are not writing to registers or memory
Single-Cycle Datapath: \texttt{addi}

- \texttt{addi} is an I-type instruction that adds the value in a register to the immediate, and writes the result to another register
- \( \text{Reg}[rt] = \text{Reg}[rs] + \text{immediate} \)
- We already have all the datapath hardware we need to implement this instruction
Single-Cycle Datapath: \( j \)

- The \( j \) instruction is a J-type instruction that writes a new value to the PC

  **J-type**

<table>
<thead>
<tr>
<th>op</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- The two least significant bits of the PC are always 00 (why?)
- The next 26 least significant bits of the new PC (PC') are taken from the jump address field (\( Instr_{25:0} \))
- Finally, the four most significant bits are taken from the value of the current PC+4
- Our current datapath cannot perform this computation to determine PC', so we need more hardware
Single-Cycle Datapath: \( j \)

- What hardware could/should we add to implement \( j \)?
Single-Cycle Datapath: $j$
Complete Single-Cycle CPU
Single-Cycle Control Unit

- The control signals are based on the opcode and funct fields of instructions, which are $Instr_{31:26}$ and $Instr_{5:0}$, respectively.
- R-type instructions use the funct field to determine the operation.

<table>
<thead>
<tr>
<th>ALUOp$_{1:0}$</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Look at funct</td>
</tr>
<tr>
<td>11</td>
<td>Not used</td>
</tr>
</tbody>
</table>
Single-Cycle Control Unit

- Since 11 is not used, we can use 1X to represent Look at \textit{funct}
- Moreover, for R-type instructions, we actually need only $\text{Funct}_{3:0}$ because the first two bits are always 10
- This simplifies the ALU decoder a little
- Next slides: control signals for the main decoder

\begin{center}
\begin{tabular}{|c|c|}
\hline
ALUOp\textsubscript{1:0} & Meaning \\
\hline
00 & Add \\
01 & Subtract \\
10 & Look at \textit{funct} \\
11 & Not used \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
ALUOp\textsubscript{1:0} & funct & ALUControl\textsubscript{2:0} \\
\hline
00 & X & 010 (Add) \\
X1 & X & 110 (Subtract) \\
1X & 100000 (add) & 010 (Add) \\
1X & 100010 (sub) & 110 (Subtract) \\
1X & 100100 (and) & 000 (And) \\
1X & 100101 (or) & 001 (Or) \\
1X & 101010 (slt) & 111 (SLT) \\
\hline
\end{tabular}
\end{center}
<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram of a computer processing unit with control logic, including signals such as Instr, RegWrite, RegDst, ALUSrc, Branch, MemWrite, MemtoReg, ALUOp<sub>1:0</sub>, and Jump.
<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp_{1:0}</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Instr.</td>
<td>RegWrite</td>
<td>RegDst</td>
<td>ALUSrc</td>
<td>Branch</td>
<td>MemWrite</td>
<td>MemtoReg</td>
<td>ALUOp1:0</td>
<td>Jump</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr.</td>
<td>RegWrite</td>
<td>RegDst</td>
<td>ALUSrc</td>
<td>Branch</td>
<td>MemWrite</td>
<td>MemtoReg</td>
<td>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</td>
<td>Jump</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>----------</td>
<td>------------------</td>
<td>------</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>Instr.</td>
<td>RegWrite</td>
<td>RegDst</td>
<td>ALUSrc</td>
<td>Branch</td>
<td>MemWrite</td>
<td>MemtoReg</td>
<td>ALUOp_{1:0}</td>
<td>Jump</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The diagram above illustrates the control flow and data flow of a RISC processor, specifically showing the instruction execution for the `addi` instruction. The control unit processes the instruction and generates control signals for various components of the processor, such as the instruction memory, ALU, and register file. The data flow involves reading from memory, performing arithmetic operations, and writing to registers. The diagram highlights the interaction between the instruction decoding, memory access, arithmetic logic unit (ALU) operations, and register file updates.
<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp1:0</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>Instr.</td>
<td>RegWrite</td>
<td>RegDst</td>
<td>ALUSrc</td>
<td>Branch</td>
<td>MemWrite</td>
<td>MemtoReg</td>
<td>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</td>
<td>Jump</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>----------</td>
<td>----------------</td>
<td>------</td>
</tr>
<tr>
<td><code>beq</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr.</td>
<td>RegWrite</td>
<td>RegDst</td>
<td>ALUSrc</td>
<td>Branch</td>
<td>MemWrite</td>
<td>MemtoReg</td>
<td>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</td>
<td>Jump</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>----------</td>
<td>----------------</td>
<td>------</td>
</tr>
<tr>
<td><code>beq</code></td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>01</td>
<td>0</td>
</tr>
</tbody>
</table>

Red = branch taken
Blue = branch not taken
<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr.</td>
<td>RegWrite</td>
<td>RegDst</td>
<td>ALUSrc</td>
<td>Branch</td>
<td>MemWrite</td>
<td>MemtoReg</td>
<td>ALUOp_{1:0}</td>
<td>Jump</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>j</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>XX</td>
<td>1</td>
</tr>
</tbody>
</table>
Datapaths: An Important Note

• We just highlighted only the part of the datapath used by the or instruction
• This doesn’t meant that the parts we did not highlight are “turned off” or made inactive somehow
• Signals are always traveling down wires, even when the CPU is doing no useful work or not “using” part of its circuitry to perform a computation
• This is why it’s important to set control signals to prevent unintended writes and other changes to the architectural state
Single-Cycle Performance

• Every instruction in the single-cycle CPU takes 1 clock cycle, even simple ones like `j`

• So, the clock period is constant and must be long enough to accommodate the slowest instruction. A MIPS program 100 lines long will take \((100 \times \text{delay\_of\_slowest\_instruction})\) picoseconds to complete.

• In most implementation technologies, the ALU, memory and register file accesses are much slower than other operations

• The \texttt{lw} instruction requires the data be read from and written to the register file and also read from memory

• This makes it one of the slowest operations, so let’s look at the critical path for the \texttt{lw} instruction
Critical Path for \( lw \) Instruction

- The critical path will necessarily start with the PC loading a new address on the rising edge of the clock
- Now, when presented with a signal, a register does not instantaneously change its state, remember – there is a delay until the register reaches a stable state and its value can be read
- A register’s clock-to-Q delay is denoted \( t_{pcd} \), so for the PC the propagation delay will be denoted \( t_{pcd_{-PC}} \)
Critical Path for \( lw \) Instruction
Critical Path for $lw$ Instruction

- The instruction memory reads the next instruction which means we incur a delay of retrieving data from memory.
Critical Path for \texttt{lw} Instruction

- Next we read \texttt{rs} from the register file and send it to \textit{SrcA}
- Meanwhile, the immediate field is sign-extended and selected at the \textit{ALUSrc} multiplexer to determine \textit{SrcB}
Critical Path for \( lw \) Instruction
Critical Path for \( lw \) Instruction

- The ALU now adds the two values to compute the effective address
Critical Path for \( lw \) Instruction

• Next the data memory reads from the effective address
Critical Path for \texttt{lw} Instruction

- Then the \texttt{MemtoReg} mux selects \textit{ReadData}
Critical Path for \( \text{lw} \) Instruction

- While all of this is happening, we compute \( PC' = PC + 4 \)
Critical Path for \texttt{lw} Instruction

- Finally, \textit{Result} must “setup” at the register file before the next rising clock edge so that it can be properly written
  - Here, “setup” refers to the fact that it takes time for the register’s value to stabilize
- Complicating the critical path delay analysis is the fact that there are two paths through the circuit we have to consider: a path through the register file (to read \texttt{rs}) and a path through the sign-extending unit and \texttt{ALUSrc} mux
  - Reading the register file will take longer, so we can safely ignore the other path
Critical Path for \( \text{lw} \) Instruction

- Therefore, the delay (CPU cycle time) is:
  \[
  T_c = t_{\text{pcq\_PC}} + t_{\text{mem}} + t_{\text{RF\_read}} + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{\text{RF\_setup}}
  \]
Critical Path for $l_w$ Instruction

• There are really different ways we could estimate the critical path delay
• For example, we have not considered the time for signals to travel down wires, but we will usually ignore this
• Also, trying to rely on a formula is not a very general way to proceed. Rather, we should determine the critical path for an instruction and then compute the delay by looking at which components lie on the datapath for that instruction.
• Suppose we have the following timings (in ps) for the combinational and memory units:

<table>
<thead>
<tr>
<th></th>
<th>I-Mem</th>
<th>Adder</th>
<th>MUX</th>
<th>ALU</th>
<th>RegFile</th>
<th>D-Mem</th>
<th>SignExt</th>
<th>Shifter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40</td>
<td>5</td>
<td>1</td>
<td>20</td>
<td>30</td>
<td>50</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Critical Path for \( lw \) Instruction

<table>
<thead>
<tr>
<th>Component</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-Mem</td>
<td>40</td>
</tr>
<tr>
<td>Adder</td>
<td>5</td>
</tr>
<tr>
<td>MUX</td>
<td>1</td>
</tr>
<tr>
<td>ALU</td>
<td>20</td>
</tr>
<tr>
<td>RegFile</td>
<td>30</td>
</tr>
<tr>
<td>D-Mem</td>
<td>50</td>
</tr>
<tr>
<td>SignExt</td>
<td>2</td>
</tr>
<tr>
<td>Shifter</td>
<td>2</td>
</tr>
</tbody>
</table>
Critical Path for $lw$ Instruction

<table>
<thead>
<tr>
<th></th>
<th>I-Mem</th>
<th>Adder</th>
<th>MUX</th>
<th>ALU</th>
<th>RegFile</th>
<th>D-Mem</th>
<th>SignExt</th>
<th>Shifter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>40</td>
<td>5</td>
<td>1</td>
<td>20</td>
<td>30</td>
<td>50</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Computing Critical Path Delays

• Let's compute the critical path delay (propagation delay) for R-type instructions in the datapath
Critical Path for R-type Instructions

You try!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>I-Mem Read</th>
<th>Adder</th>
<th>MUX</th>
<th>ALU</th>
<th>RegFile Read</th>
<th>RegFile Write</th>
<th>D-Mem Read</th>
<th>D-Mem Write</th>
<th>SignExt</th>
<th>Shifter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40</td>
<td>5</td>
<td>3</td>
<td>20</td>
<td>30</td>
<td>45</td>
<td>50</td>
<td>40</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
Critical Path for R-type Instructions

\[ t = 0 \]

I-Mem Read | Adder | MUX | ALU | RegFile Read | RegFile Write | D-Mem Read | D-Mem Write | SignExt | Shifter
---|---|---|---|---|---|---|---|---|---
40 | 5 | 3 | 20 | 30 | 45 | 50 | 40 | 4 | 6
Rules for Adding Instructions

• Determine which group of instructions the new instruction is most like, if any.
• Determine the flow of the data from the Fetch stage through the datapath.
• Determine any necessary changes to the existing datapath and any additional hardware that may be required (always add hardware as a last resort).
• Determine the control signals for the existing system and added hardware.
• You may need to change existing controls to add more bits.
• Make sure you change the control values for the existing instructions and specify any new controls.
• Don’t break existing functionality!
Single-Cycle Datapath: \texttt{swi}

- Suppose we invent a new R-type instruction, \texttt{swi}, that performs the following operation:
  \[ \text{Mem[Reg[rd] + Reg[rs]]} = \text{Reg[rt]} \]

- \texttt{swi} would read two registers and add their contents to determine the effective address of the location in memory to store the contents of \texttt{rt}

- In a single cycle we can use each component of the datapath only once
  - For example, this means we cannot read from the register file on two separate occasions in a single cycle
  - Therefore, we will need to add another read port to the register file (\texttt{RD3}) and also a corresponding address input (\texttt{A4})
Single-Cycle Datapath: \textit{swi}

\[ \text{Mem[Reg[rd] + Reg[rs]]} = \text{Reg[rt]} \]
Single-Cycle Datapath: \textit{swi}

\[ \text{Mem}[	ext{Reg}[rd] + \text{Reg}[rs]] = \text{Reg}[rt] \]
### Single-Cycle Datapath: \texttt{swi}

<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp_{1:0}</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>xx</td>
<td>1</td>
</tr>
<tr>
<td>swi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Single-Cycle Datapath: `swi`

<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp_{1:0}</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>X</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>0</td>
<td>X</td>
<td>XX</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>XX</td>
<td>1</td>
</tr>
<tr>
<td>swi</td>
<td>0</td>
<td>X</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: \texttt{swinc}

• Imagine an I-type instruction that could store a word and increment a register by 4 in one instruction (i.e., one clock cycle)

• In other words:
  \[
  \text{Mem}[	ext{Reg[rs]} + \text{immediate}] = \text{Reg[rt]}
  \]
  \[
  \text{Reg[rs]} = \text{Reg[rs]} + 4
  \]

• Format: \texttt{swinc \texttt{rt, immediate(rs)}}

• The \texttt{sw} portion of the instruction is unchanged. No datapath modifications are required for this half of the instruction.

• For the second half of the instruction, the value in the \texttt{rs} register must be incremented by 4. How could we do this? Let’s look over the datapath diagram.
Single-Cycle Datapath: $\text{swinc}$

Mem[Reg[rs]+immediate] = Reg[rt]
Reg[rs] = Reg[rs] + 4
Single-Cycle Datapath: \texttt{swinc}

- All adders are being used by other components, so we need to insert another adder.
- This new adder will take the \texttt{RD1} output and the immediate value 4 as its inputs, and output the sum of those two values.
- This sum needs to be written to the register file.
- Now, the \textit{MemtoReg} mux determine what data is sent to the write port (\texttt{WD3}) of the register file.
- We will need to add a third input to that mux so we can send the output of our new adder to the register file.
- Also, we need to provide the correct register number to \texttt{A3}, so we will need to add a third input to the \textit{RegDst} mux.
Single-Cycle Datapath: \texttt{swinc}

Mark the datapath for the instruction and give the control signals.

\begin{center}
\begin{tikzpicture}
\end{tikzpicture}
\end{center}

\begin{equation}
\text{Mem[Reg[rs]+immediate]} = \text{Reg[rt]}
\end{equation}

\begin{equation}
\text{Reg[rs]} = \text{Reg[rs]} + 4
\end{equation}
Single-Cycle Datapath: \texttt{swinc}

Mark the datapath for the instruction and give the control signals.

\[ \text{Mem[Reg[rs]+immediate]} = \text{Reg[rt]} \]
\[ \text{Reg[rs]} = \text{Reg[rs]} + 4 \]
Single-Cycle Datapath: \texttt{swinc}

Mark the datapath for the instruction and give the control signals.

\[
\text{Mem[Reg[rs] + immediate]} = \text{Reg[rt]}
\]

\[
\text{Reg[rs]} = \text{Reg[rs]} + 4
\]
Single-Cycle Datapath: \texttt{bnmul4}

- Imagine a MIPS instruction that would take a branch if the value in \texttt{rs} were not a multiple of 4: \texttt{bnmul4 rs, label}

- If \texttt{Reg[rs]} is NOT a multiple of 4 then
  \[ \text{PC'} = \text{PC} + 4 + \text{SignImm} \times 4 \]
  \[ \text{else} \]
  \[ \text{PC'} = \text{PC} + 4 \]

- How could we implement this? What hardware and logic would we need to add to the single-cycle datapath and control unit?

- The binary representation of an integer (positive or negative) that is a multiple of four will end in 00

- So we will branch if either or both of the two least significant bits are 1s:
  \[ \text{Reg[rs][0]} = 1 \text{ or } \text{Reg[rs][1]} = 1 \]
Single-Cycle Datapath: \texttt{bnmul4}

Modify the datapath and control unit, mark the datapath for the instruction and give the control signals for the new instruction.

\begin{quote}
\textbf{if} Reg[rs] \textbf{is NOT a multiple of 4} then
\begin{align*}
\text{PC'} &= \text{PC} + 4 + \text{SignImm} \times 4 \\
\text{else} & \\
\text{PC}' &= \text{PC} + 4
\end{align*}
\end{quote}
Single-Cycle Datapath: \( \text{bnmul4} \)

Modify the datapath and control unit, mark the datapath for the instruction and give the control signals for the new instruction.

\[
\text{if Reg[rs] is NOT a multiple of 4 then} \quad \text{PC'} = \text{PC} + 4 + \text{SignImm} \times 4 \\
\text{else} \quad \text{PC'} = \text{PC} + 4
\]
Single-Cycle Datapath: \texttt{bnmul4}

Modify the datapath and control unit, mark the datapath for the instruction and give the control signals for the new instruction.

\begin{equation*}
\text{if Reg[rs] is NOT a multiple of 4 then} \quad PC' = PC + 4 + \text{SignImm} \times 4 \\
\text{else} \quad PC' = PC + 4
\end{equation*}

Red = branch taken
Single-Cycle Datapath: \texttt{jrimm}

- Implement the instruction \texttt{jrimm}, which changes the program counter as indicated in the pseudocode below

\begin{verbatim}
# "jump register immediate": jrimm $rs, $rt, immed
if Reg[$rs] < SignImm then
    PC' = PC + 4 + Reg[$rt] \text{ <<} 2
else
    PC' = PC + 4 + SignImm \text{ <<} 2
\end{verbatim}
Single-Cycle Datapath: \( j \text{rimm} \)

Modify the datapath and control unit, and update the table of control signals as needed.

if \( \text{Reg}[$rs] < \text{SignImm} \) then

\[
\text{PC}' = \text{PC} + 4 + \text{Reg}[$rt] \ll 2
\]

else

\[
\text{PC}' = \text{PC} + 4 + \text{SignImm} \ll 2
\]
Single-Cycle Datapath: \texttt{jr imm}

Modify the datapath and control unit, and update the table of control signals as needed.

\[
\text{if } \text{Reg[rs]} < \text{SignImm} \text{ then} \\
\text{PC'} = \text{PC} + 4 + \text{Reg[rt]} \ll 2 \\
\text{else} \\
\text{PC'} = \text{PC} + 4 + \text{SignImm} \ll 2
\]
### Single-Cycle Datapath: \texttt{jrimm}

<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp_{1:0}</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>xx</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{jrimm}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: \textit{jrimm}

<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp\textsubscript{1:0}</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>\textit{lw}</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>\textit{sw}</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>\textit{beq}</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>\textit{addi}</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>\textit{j}</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>XX</td>
<td>01</td>
</tr>
<tr>
<td>\textit{jrimm}</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>10</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: \textit{jr-imm}

Modify the datapath and control unit, and update the table of control signals as needed.

\textbf{if} \ Reg[$rs] < \text{SignImm} \ \textbf{then} \\
\hspace{1em} \text{PC'} = \text{PC} + 4 + \text{Reg}[$rt] \land \text{SignImm} \land 2 \\
\textbf{else} \\
\hspace{1em} \text{PC'} = \text{PC} + 4 + \text{SignImm} \land 2
Single-Cycle Datapath: $jrimm$

Modify the datapath and control unit and update the table of control signals as needed.

if $\text{Reg}[$rs$] < \text{SignImm}$ then
    $PC' = PC + 4 + \text{Reg}[$rt$] << 2$
else
    $PC' = PC + 4 + \text{SignImm} << 2$
Single-Cycle Datapath: \textit{jrimm}

<table>
<thead>
<tr>
<th>Instr</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp$_{1:0}$</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>\textit{j}</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>\textit{xx}</td>
<td>1</td>
</tr>
</tbody>
</table>

\textit{jrimm}
## Single-Cycle Datapath: \textit{jrimm}

<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp_{1:0}</th>
<th>Jump</th>
<th>jrimm</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>00</td>
<td>X</td>
</tr>
<tr>
<td>lw</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>00</td>
<td>X</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>00</td>
<td>X</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>01</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>X</td>
</tr>
<tr>
<td>j</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>x</td>
<td>XX</td>
<td>01</td>
<td>X</td>
</tr>
<tr>
<td>jrimm</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>