CSE 220:
System Fundamentals I
Unit 15:
MIPS Architecture:
Multicycle Processors
Multicycle Processors

• The single-cycle processor has three major weaknesses:
  1. The clock cycle must be long enough to support the slowest instruction (\texttt{lw}), but most operations are faster
  2. It requires three adders, which are expensive
  3. It has separate data and instruction memories, which is not realistic

• A multicycle CPU addresses these weaknesses by breaking an instruction into multiple, shorter steps or stages, each of which requires about the same amount of time (1 cycle)

• Each instruction will execute only those stages that are actually required. (So, \textbf{1 stage = 1 cycle}. This is key!) In the single-cycle CPU, these shorter instructions simply wait for the unused stages to complete, which wastes time.
Multicycle Processors

• In a multicycle CPU, simpler instructions should run faster
• Also, the multicycle CPU reuses expensive hardware on multiple cycles so we don’t need to have duplicate hardware units (e.g., adders)
• But, in a given stage, how do we know which stages apply, and therefore, which parts of the datapath we need to use?
  • Answer: we need to add extra control signals and control hardware, which complicates things.
• Conversely, these control signals will also let us use the components for different purposes during different stages.
  • e.g., the ALU can be used to compute PC+4, or an effective memory address (base + offset), or a branch address. We will also have a single data/instruction memory.
The Multicycle Approach

• Break up instructions into stages, with each stage taking one clock cycle

• Restrict each cycle to use only one major functional unit (e.g., ALU, memory)

• Each functional unit is used only once per cycle, but perhaps several times per instruction over several cycles

• At the end of each cycle must store intermediate values (e.g., the output of the ALU) for use in later cycles

• We do so by introducing **non-architectural state elements**. These are “internal” registers that are not part of instructions and not visible to the programmer

• Let’s go ahead now and build the multicycle MIPS process from the ground-up
Multicycle State Elements

- PC and register file are the same as in the single-cycle CPU
- But now we have a unified instruction/data memory
- We will be able to read the instruction in one cycle and read or write data in a separate cycle
- We’ll work out the `lw` instruction datapath first
Instruction Fetch Datapath

- The Instruction Register (IR) is a new non-architectural element that holds the instruction for use in later cycles.
- We have a control signal, IRWrite, which is asserted when the IR should be updated with a new instruction.
Multicycle Datapath: $\text{l}w$

- Now we need the base address from the source register, in $\text{rs}$, which is given as $\text{Instr}_{25:21}$ in the instruction.
- These 5 bits are sent to input $A1$. The register file reads the address in $\text{rs}$ onto $RD1$. This value is stored in non-architectural register $A$. 
Multicycle Datapath: \texttt{lw}

- \texttt{lw} requires an offset, given as $Instr_{15:0}$, which must be sign-extended. A non-architectural register is not needed because the instruction will not change during execution.
Multicycle Datapath: \texttt{l W}

- The address of the word to load is $\text{SignImm} + \text{Reg}[rs]$
- We set $\text{ALUControl}_{2:0} = 010$ to perform this addition
- $\text{ALUResult}$ is saved in non-architectural register $\text{ALUOut}$
- We now have our effective address, so we are almost done
- Next step: read data from the memory at that address
Multicycle Datapath: \( lw \)

- We add a MUX controlled by \( IorD \) that indicates either an instruction address or data address (\( Adr \))
- The data read from memory is stored in non-architectural register \( Data \) (called \( MDR \), Memory Data Register, in the Hennessy & Patterson book)
Multicycle Datapath: \( lw \)

- The address mux (\( IorD \)) lets us reuse the memory during a \( lw \)
- In the first step, \( IorD = 0 \) and the address is taken from the PC
- In the second step, \( IorD = 1 \) and we take the address computed by the ALU (as \( ALUOut \))
- This is why we no longer need separate instruction and data memories
• Finally, the data is written back into the register file in register $rt$, which is given by $Instr_{20:16}$
Multicycle Datapath: lw

- We also need to update the PC to add 4 to it
- Provided that we can find some time (i.e., a cycle) when the ALU is not in use, we can use the ALU to calculate PC+4
- We add a mux in front of SrcA to choose between A and the PC
Multicycle Datapath: \texttt{lw}

- We also add a mux in front of \texttt{SrcB} to choose between \texttt{SignImm} and the constant 4 (the other 2 mux inputs will be used in other instructions later on)
- We assert \texttt{PCWrite} to update the value of PC
- This concludes \texttt{lw}. Now let’s look at \texttt{sw}. 
Multicycle Datapath: $S_W$

- We know that $rt$ tells us which register has the contents to write to memory
- We add a non-architectural register $B$ to hold it
- $\text{MemWrite}$ is asserted to write the register’s contents at the address calculated by the ALU
Multicycle Datapath: R-type

- To support R-type instructions we need to be able to read rs and rt out of the register file, perform an ALU operation, and write the result to rd
- We add two muxes: MemtoReg selects whether WD3 comes from ALUOut (for R-type) or from Data (for lw)
Multicycle Datapath: R-type

- The other mux, \textit{RegDst}, selects whether the destination register comes from \textit{rd} (for R-type) or \textit{rt} (for \texttt{lw})
- That’s all we need for R-type instructions, so now on to \texttt{beq}
Multicycle Datapath: \texttt{beq}

- To execute \texttt{beq} we need to do \texttt{rs}–\texttt{rt} and check the \texttt{Zero} flag to see if the difference is 0.
- We also need to calculate \texttt{PC}+4 and then add to that sum the value of \texttt{SignImm} \times 4. So two additions are needed.
- The ALU can do the additions on different cycles.
Multicycle Datapath: \texttt{beq}

- On one step, $ALUSrcA = 0$ to select the PC and $ALUSrcB = 01$ to select 4. PC+4 is written back to the PC.
- On the next step, $ALUSrcA = 0$ to select the new PC value and $ALUSrcB = 11$ to select the shifted, sign-extended immediate. The sum is sent to $ALUOut$. 
Multicycle Datapath: \texttt{beq}

- Note that for the first step $\text{PCSrc} = 0$ and on the second step $\text{PCSrc} = 1$
- This $\text{PCSrc}$ mux controls what is written to the PC
- The $\text{Branch}$ signal is asserted to indicate we are performing a branch. $\text{PCEn}$ will be 1 if both $\text{Zero}$ and $\text{Branch}$ are 1.
“Complete” Multicycle Processor

- This datapath does not implement the j instruction
Five CPU Stages

- There are five fundamental stages the multicycle CPU executes (each stage is 1 cycle long):
  1. Fetch the instruction and compute $PC' = PC + 4$ (all instructions require this)
  2. Decode the instruction and fetch registers (this is also an instruction-independent stage)
  3. Execute ALU operation (instruction-dependent)
  4. Memory access (lw or sw) or complete R-type instruction
  5. Write to register file (lw only)

- Not every instruction requires all five stages. Instructions will vary from 3-5 cycles per instruction.
Stage 1: Instruction Fetch

- Use the PC to retrieve the instruction from the Memory and store the 32-bit instruction in the Instruction Register
- Increment the PC by 4 and store the result back into the PC
Stage 2: Instruction Decode / Register Fetch

- This stage has three components:
  - Use the opcode to determine which instruction this is (control unit)
  - Read registers $rs$ and $rt$ from the register file (in case we need them)
  - Compute the branch address (in case the instruction is a branch): $ALUOut = PC + SignImm \times 4$. Note that by this Stage we have already added 4 to the PC, so we are actually computing $(PC + 4) + SignImm \times 4$
- The control lines are not set on the instruction type yet (we are busy “decoding” it)

- The marked datapath shows both the register fetch and branch addr. computation $ALUOut = PC + SignImm \times 4$
Stage 3: Execute ALU Operation

- This stage is instruction-dependent
- The ALU is performing one of four functions, based on instruction type (memory, R-type, branch, jump)
- Note that we have not yet implemented the j instruction in the datapath. We will do that later. So for now we will look at the first three instruction types only.
Stage 3: Execute ALU Operation

Memory reference/immediate instructions (\texttt{lw}, \texttt{sw}, \texttt{addi}, etc.): \texttt{ALUOut} = A + \text{SignImm}
Stage 3: Execute ALU Operation

R-type (add, sub, slt, etc.): $ALUOut = A \ op B$
Stage 3: Execute ALU Operation

Branch (beq): if (A equals B) then $PC = ALUOut$
Stage 4: Memory Access ($lw$, $sw$) or R-type Completion

- This stage is required only for R-type instructions or memory-access ($lw$, $sw$)
- These instructions are writing data to somewhere during Stage 4:
  - $lw$ writes to the Data register
  - $sw$ writes to main memory
  - R-type instructions write to the register file
Stage 4: Memory Access ($l^w$)

Data = Mem[ALUOut]
Stage 4: Memory Access (sw)

\[ \text{Mem}[\text{ALUOut}] = B \]
Stage 4: Write Reg. File (R-type)

$\text{Reg}[IR[15:11] = ALUOut}$
Stage 5: Mem. Writeback ($lw$ only)

- Write the data from memory into the registers
- $\text{Reg}[\text{IR}[20:16]] = \text{Data}$
## Multicycle Implementation Summary

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Action for R-type Instructions</th>
<th>Action for Memory Reference Instructions</th>
<th>Action for Branches</th>
<th>Action for Jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Instruction Fetch</td>
<td></td>
<td>$IR = Memory[PC]$</td>
<td>$PC' = PC + 4$</td>
<td></td>
</tr>
<tr>
<td>2. Instruction Decode / Register Fetch</td>
<td></td>
<td>$A = Reg[IR[25:21]]$</td>
<td>$B = Reg[IR[20:16]]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$ALUOut = PC + (SignImm + IR[15:0]) \ll 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Exec./Addr. Computation, Branch/Jump complete</td>
<td>$ALUOut = A \text{ op } B$</td>
<td>$ALUOut = A + \text{SignExtend}(IR[15:0])$</td>
<td>if $(A \text{ eq } B)$ then $PC' = ALUOut$</td>
<td>$PC' = PC[31:28], IR[25:0] \ll 2$</td>
</tr>
<tr>
<td>5. Mem. read completion</td>
<td></td>
<td>$lw: Reg[IR[20:16]] = Data$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multicycle Implementation Summary

• The chart on the previous slide summarizes all of the different operations which occur in each stage.

• Remember that each stage is a single clock cycle. The clock cycle time is dependent on the critical path for any of the different steps.

• Each instruction type requires a different number of clock cycles to complete, based on the number of stages required.
  • R-type: 4 clock cycles
  • lw: 5 clock cycles
  • sw: 4 clock cycles
  • beq: 3 clock cycles
  • j: 3 clock cycles
Multicycle Implementation Summary

• The length of the clock cycle is determined by the length of the longest critical path
• The critical path must be calculated for each of these stages/steps separately.
• Remember: for any clock cycle, the processor does not know which stage/step or the instruction it is executing, so the clock has to operate for the worst case timing
Multicycle Timing Example

• Consider the following MIPS code:
  
  ```
  lw $t2, 0($t3)  
  lw $t3, 4($t3)  
  beq $t2, $t3, Label  # assume not taken  
  add $t5, $t2, $t3  
  sw $t5, 8($t3)  
  ```

• How many cycles will it take to execute this code?
  • Answer: 5 + 5 + 3 + 4 + 4 = 21 cycles

• What is going on during the 8th cycle of execution?
  • Answer: 3rd stage of `lw $t3, 4($t3)`
  • Stage 3 of `lw` uses ALU to calculate the effective memory address. So it’s $ALUOut = Reg[$t3] + SignExtend(4)$
Multicycle Timing Example

• Consider the following MIPS code:
  
  \texttt{lw \$t2, 0($t3)}
  \texttt{lw \$t3, 4($t3)}
  \texttt{beq \$t2, \$t3, Label} \quad \# \text{assume not taken}
  \texttt{add \$t5, \$t2, \$t3}
  \texttt{sw \$t5, 8($t3)}

• In what cycle does the actual addition of $t2$ and $t3$ take place?

• Answer: the add starts after $5 + 5 + 3 = 13$ cycles

• The ALU computes the sum of $t2$ and $t3$ in stage 3 of the operation: $13 + 3 = 16$. So the add happens in cycle #16
Multicycle Control

- As with the single-cycle processor, the control unit is partitioned into a main controller and an ALU decoder.
- The main controller is a finite state machine that applies the proper control signals on the proper cycles (stages).
- The sequence of control signals depends on the instruction being executed.
Finite State Machines

- A **finite state machine** (FSM) is a circuit that can be in one of a finite number of states at any given time.
- An FSM has $M$ inputs, $N$ outputs and $k$ bits of state, meaning that the FSM has $2^k$ possible states it can be in.
- It consists of two blocks of combinational logic:
  - the **next state logic**, which computes the next state as a function of the current state and inputs.
  - the **output logic**, which determines the circuit output as the FSM transitions from the current state to the next.
- In the context of the multicycle CPU, a FSM will determine which control signals to assert based on the *instruction* being executed and the current instruction *stage* that is being executed.
Finite State Machine Example

• Imagine that we need to invent a controller for a traffic light at a busy intersection on a university campus
  • Traffic sensors: $T_A$, $T_B$ (TRUE when there’s traffic)
  • Lights: $L_A$, $L_B$ (red, yellow or green)
• Every 5 seconds the lights may change, based on the traffic sensor
• While traffic is present on a road with a green light, the lights will not change
• The lights will change once that traffic stops so as to give the other traffic a chance to move through the intersection
Finite State Machine Example

• Here we have a black box view of the FSM
• As with our CPU, this system has a clock (with a clock period of 5 seconds)
• To model the behavior of this circuit we can draw a state transition diagram
Finite State Machine Example

- In a state transition diagram, circles represent states and arcs represent transitions between states.
- If an arc has a label, the label indicates what input triggers the transition.
- For example, if the FSM is in state S2 and no traffic is detected by sensor $T_B$, the FSM transitions to state S3 and changes $L_A$ to red and $L_B$ to yellow.
- Outputs are listed within each state.
Multicycle Controller FSM

• The ALU decoder is the same as before, so we need to design a FSM for the Main Controller to output the (i) multiplexor select signals, (ii) Write Enable signals and (iii) Register Enable signals

• Only relevant control signals will be listed as outputs in each FSM state. Unlisted select signals are don’t-cares.

• Enable signals will be listed only when they are asserted. Otherwise, they are 0 (not don’t-care).
FSM State S0 (Fetch)

- Every instruction starts with a fetch of the instruction itself
FSM State S1 (Decode)

- In S1, registers rs and rt are read from the register file
- opcode and funct are sent to the control unit
- The FSM waits one cycle for reading and decoding to complete
FSM State S2 (MemAdr): \texttt{lw} / \texttt{sw}

- Depending on the \texttt{opcode}, the FSM transitions into one of several possible states
- If the instruction is \texttt{lw} or \texttt{sw}, the FSM moves to state S2
- In S2, the CPU computes the address by adding the base address to the sign-extended immediate
FSM State S2 (MemAdr): \texttt{lw} / \texttt{sw}

- In S2, the CPU computes the address by adding the base address to the sign-extended immediate
Multicycle Controller FSM: \texttt{lw}

- From S2, when \texttt{opcode} is \texttt{lw}, the CPU must read data from memory and write it to the register file.
- In S3 this memory read takes place.
- In S4 the word from memory is written to the register file.
FSM State S3 (MemRead): \texttt{lw}

- Read the word from memory at the effective address stored in \textit{ALUOut} and write it into the memory data register (\textit{Data})
FSM State S4 (Mem Writeback): \( lw \)

- Write the word in the memory data register \((Data)\) into the register file in \( rt \)
Multicycle Controller FSM: \textit{SW}

- From S2, when \textbf{opcode} is \textit{sw}, the CPU must write a word in the register file to memory.
- In S5 this memory write takes place.
FSM State S5 (MemWrite): $s^w$

- The contents of $rt$ are written to the effective address stored in $ALUOut$.
FSM State S6 (Execute): R-type

- If in S1 the instruction is an R-type instruction, the FSM moves to state S6 to execute the instruction
- Then it goes to S7 to write the result to the register file
FSM State S6 (Execute): R-type

- Read registers $A$ and $B$, perform the ALU operation and store the result in the $ALUOut$ register

Op = R-type

$S6: \text{Execute}$

$ALUSrcA = 1$
$ALUSrcB = 00$
$ALUOp = 10$
FSM State S7 (ALU Writeback): R-type

- Write the contents of the ALUOut register to register \( rd \)
**FSM State S8 (Branch): beq**

- If in S1 the instruction is a branch instruction, the FSM moves to S8.
- However, a branch requires a few calculations. How do we handle them?
FSM State S8 (Branch): \textbf{beq}

- The \textbf{beq} instruction needs the ALU twice: first to compute $PC + 4 + \text{SignImm} \times 4$ and then to compute $rs - rt$
- $PC + 4$ is computed in Cycle 1, which corresponds to State S0 (Fetch)
- Currently, State S1 (Cycle 2) doesn’t use the ALU, so let’s do the computation of $PC + \text{SignImm} \times 4$ then. Note by now, $PC$ has already been incremented by 4.
FSM State S8 (Branch): \texttt{beq}

- Once the FSM reaches State S8 the CPU has computed the branch address, regardless of whether the branch will be taken or not.
- Now in State S8 the CPU determines if the branch should be taken or not by calculating $rs - rt$ and checking the Zero flag.
FSM State S8 (Branch): \texttt{beq}

- Note here that \textit{ALUOut} contains the branch address (computed in the previous CPU cycle), not the output of the ALU.
- This helps underscore the need for these non-architectural registers.
Adding Instructions

• Modifications to the multicycle datapath must be considered in stages
• Any new instructions should be broken into sub-tasks/sub-stages which are similar to the existing stages
• Whenever possible, reuse the existing datapath stages and controller states
• Modification to the datapath (new data lines, multiplexers, etc.) should be added only when necessary
• Large components, such as ALUs, memories, registers, etc. should be added only as a last resort
Adding Instructions

• Care must be made not to increase the critical path of any stage
• Adding an extra clock cycle is preferred over increasing the critical path of a stage
• Why? Extra stages only affect that instruction type, rather than all instructions, which *would* be affected by increasing the critical path in a stage.
Adding Instructions: \texttt{addi}

- The datapath includes all the hardware to execute operations like \texttt{addi}, so what remains is to modify the FSM accordingly.
Adding Instructions: \texttt{addi}

- Let’s look at the datapath. This is the version that does not have the hardware and control signals for jumps.
Adding Instructions: \texttt{addi}

- \texttt{addi} is an I-type instruction that writes to the register file. Similar to the \texttt{lw} instruction, but has no memory access.
Adding Instructions: addi

- How should we modify the FSM?
- States S0 and S1 are unchanged
- In Cycle 3 we should be able to add rs and the sign-extended immediate
  - $ALUSrcA = 1$
  - $ALUSrcB = 10$
  - $ALUOp = 00$
  - $ALUCtrl = 010$
Adding Instructions: addi
Adding Instructions: addi

• Now we need to write the sum to the register file
• That sum is sitting in the ALUOut register, so we need to go ahead and select that value to be saved in register rt
• We can look to State S7 (ALU Writeback) for some inspiration
  • RegDst = 0
  • MemtoReg = 0
  • RegWrite = 1
Adding Instructions: `addi`
Adding Instructions: \( j \)

- We have seen the datapath with the \( j \) hardware but haven’t discussed it yet. Here’s what we have without it:
Adding Instructions:  \( j \)

- Recall that \( j \) takes the 4 most significant bits of the PC and prepends them to the 28 bits formed by left-shifting the instruction’s immediate value by 2 bits.
- An additional input and control bit are added to \( PCSrc \) to support the sending of the jump address to the PC.
Adding Instructions: \(j\)

- In State S0 (Cycle 1) the instruction is fetched, and in State S1 (Cycle 2) bits \(IR_{25:21}\) and \(IR_{20:16}\) are read into the register file.

- These S1 actions are not applicable to jumps, but it happens anyway and there’s nothing we can do about it.

- Also, during S1 the CPU is calculating a branch address (and thus, reading the PC). Again, it doesn’t help with a jump, but that’s OK because we won’t use the result.

- During Cycle 2 (State S1) the jump target will be assembled from \(PC_{31:28}\) and the right-shifted immediate.

- We will add a new state, State S11 (Cycle 3) during which the PC is updated with the jump target.

- \(PCWrite = 1\) and \(PCSsrc = 10\)
Adding Instructions: j
Multicycle CPU Performance

• The main motivation for developing the multicycle CPU to replace the single-cycle CPU was to avoid making all instructions take as long as the slowest one (usually **lw**)

• The multicycle CPU does less work in one cycle than the single-cycle CPU because each instruction has been broken into multiple, shorter steps

• In the worst case, for the multicycle CPU an instruction will take 5 cycles

• Now, if each of those cycles is at most one-fifth as long as a cycle for the single-cycle CPU, then we are ahead

• However, it turns out we probably cannot achieve higher performance with the multicycle CPU
Multicycle CPU Performance

• There are a couple of reasons. For the single-cycle CPU we had to pay the register “setup” delay once per instruction when updating the value in the PC.

• With the multicycle CPU we pay this penalty in every single stage (for the PC and the non-architectural registers)

• Moreover, the CPU’s clock cycle for the multicycle CPU is going to be limited by the slowest stage, which is typically going to be a stage that involves accessing memory

• We have gone through all this work in the hopes of making a faster CPU, but the multicycle CPU is actually going to be slower in many circumstances than the single-cycle CPU

• It might be cheaper, though, because it has fewer components than the single-cycle CPU
Adding Instructions: jr

• Jump to the address specified by a register:
  • PC = Reg[rs]

• This instruction could use either the R or I format, as it only requires one register to be specified

• There are multiple ways to add this instruction to the datapath and control. Depending on our goal (e.g., minimize hardware changes, minimize instruction time) we will choose one option over others.

• We will look at three approaches:
  1. Modify datapath (add hardware) and FSM
  2. Don’t modify datapath, but change FSM
  3. Don’t modify datapath, but change FSM in a better way than approach #2
Adding Instructions: \texttt{jr} (Ver. 1)

- Approach #1: modify the datapath
- Fetch and decode are performed as normal
- At this point, \texttt{Reg[rs]} is in the A register of the datapath
- We need to send A’s contents to the PC. This would be in the third clock cycle of the instruction.
Adding Instructions: jr (Ver. 1)

- The *PCSrc* mux controls what is sent to the PC, but none of the inputs is connected to *A*, so we will need to modify it by adding a fourth input.
- Finally, we need to add a new state to the FSM that will direct the control unit to output the right control signals.
Adding Instructions: \( jr \) (Ver. 1)
Adding Instructions: $jr$ (Ver. 2)

- Approach #2: no datapath modification (the long way)
- Fetch and decode are performed as normal
- At this point, $\text{Reg}[rs]$ is in the $A$ register of the datapath
- Of the three inputs to $PCSrc$, which one should we use?
- Only 00 or 01 would make sense
Adding Instructions: \texttt{jr} (Ver. 2)

- In either case we need to send \texttt{Reg[rs]} through the ALU
- Then the question becomes whether we read \texttt{ALUResult} directly or send it the \texttt{ALUOut} register and then read it out on the following clock cycle
- Let’s see what happens if we send it to \texttt{ALUOut}
Adding Instructions: \( jr \) (Ver. 2)

- In Stage 3 of the instruction we can pass \( A \) through the ALU by adding/subtracting with 0
- Where does the 0 come from? In both the R-type and I-type instruction formats a second register can be specified.
- If we assume the assembler places the \$0 \) register (000000) in the instruction’s \( rt \) field, then in the decode stage the value zero will be placed in register \( B \)
- If we use the R-type instruction format, we can use the \( funct \) field to specify the addition or subtraction for \textit{ALUControl}
- In this way we can use the existing Stage 3 for R-type instructions
Adding Instructions: \textit{jr} (Ver. 2)

- The result of the calculation is stored in $ALUOut$ (red).
- In the next cycle (Stage 4), $PC=ALUOut$ (blue). This requires a new state for the FSM.
Adding Instructions: \texttt{jr} (Ver. 2)

PCSrc = 01
PCWrite

or \texttt{Op = JR}
Adding Instructions: \( \text{j} \text{r} \) (Ver. 3)

- Approach #3: no datapath modification (the short way)
- Fetch and decode are performed as normal
- At this point, \( \text{Reg}[\text{rs}] \) is in the \( A \) register of the datapath
- On the third clock cycle (Stage 3) we will read the \( \text{ALUResult} \) directly and send it to the PC
Adding Instructions: \texttt{jr} (Ver. 3)

- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 00
- PCSrc = 00
- PCWrite

Op = JR
Adding Instructions: \texttt{swinc}

- This fictional I-type instruction stores the contents of \texttt{Reg[rt]} at memory address \texttt{Reg[rs]+SignImm}
- It also adds 4 to \texttt{rs} and writes the result back to \texttt{rs}
- \texttt{Mem[Reg[rs]+SignImm] = Reg[rt]}
  \texttt{Reg[rs] = Reg[rs] + 4}
- Format: \texttt{swinc rt, immediate(rs)}
- What we have is a typical \texttt{sw} instruction with an extra increment by 4 of a register
- Based on this, we should be considering the stages and control flow for the \texttt{sw} instruction
- Fetch and decode are performed as normal
- Stage 3 takes care of computing \texttt{Reg[rs]+SignImm}
Adding Instructions: \texttt{swinc}

- Stage 4 of \texttt{sw} (State S5) does not use the ALU, so this means we can use it during that state to compute \( \text{Reg}[rs] = \text{Reg}[rs] + 4 \)

- However, we have a potential issue. In State S5, if we look at the FSM and match it up with the datapath figure, what is actually going on?
  - \( \text{Mem}[\text{ALUOut}] = \text{Reg}[rt] \)
  - Uh oh – calculating \( \text{Reg}[rs] + 4 \) will require us to write to the \text{ALUOut} register, meaning we need to read and write \text{ALUOut} in the same cycle
  - \textbf{Fortunately, this is not a problem because register reads take place at the start of a cycle and register writes take place at the end of a cycle.}
Adding Instructions: \texttt{swinc}

• So what will happen is this: at the start of Stage 4 the effective memory address computed during Stage 3 will be read out of $ALUOut$ and sent to the $Adr$ input of Memory

• Meanwhile, the ALU will compute $\text{Reg}[rs]+4$

• At the end of Stage 4, that sum will be written to $ALUOut$

• A new stage must be added to perform $\text{Reg}[rs] = ALUOut$

• To do this, the $RegDst$ mux must be modified to take the $rs$ register as an input

• So, to summarize, we need to modify State S5 to instruct the ALU to perform the addition and we need to add a new state to cause that sum to be written to the register file
Adding Instructions: $swinc$

- Modified Stage 4 of $sw$
- Red indicates the regular activity of $sw$
- Blue indicates the new activity to compute
  $$ALUOut = Reg[rs] + 4$$
Adding Instructions: \texttt{swinc}

- New Stage 5 to finish executing \texttt{swinc}
Adding Instructions: \texttt{swinc}

\begin{itemize}
\item \texttt{Op = SWINC}
\item \texttt{ALUSrcA = 1}
\item \texttt{ALUSrcB = 01}
\item \texttt{ALUOp = 00}
\end{itemize}

\begin{itemize}
\item \texttt{RegDst = 10}
\item \texttt{MemtoReg = 0}
\item \texttt{RegWrite}
\end{itemize}