CSE 220: System Fundamentals I
Unit 16: MIPS Architecture: Pipelined Processors
Pipelining Basics

• What we have seen so far in the single-cycle and multicycle CPUs is a very simplified approach to the execution of MIPS instructions
  • We fetch an instruction, decode it, and execute it completely before starting another instruction
  • Only one instruction is handled at a time by the CPU
  • A major negative (for both microarchitectures) is that the datapath has several parts/components which sit idle when other parts are in use
  • In the multicycle machine we tried to mitigate this problem by combining units (single ALU, one memory for data and instructions). We also tried to do as much work as possible in each state.
Parallelism

• There are two fundamental ways of performing two or more operations in parallel

• **Spatial parallelism:** duplicate hardware performs multiple tasks at once
  • Example: a baker bakes two cakes at a time, each in its own oven

• **Temporal parallelism:** a single task is broken into multiple stages
  • Also called **pipelining**
  • Example: in a car factory, multiple cars are being assembled at once, but each car in the pipeline is at a different stage of assembly
Spatial vs. Temporal Parallelism

- Spatial and temporal parallelism in a cookie bakery

Spatial Parallelism

- Tray 1: Ben 1
- Tray 2: Alyssa 1
- Tray 3: Ben 2
- Tray 4: Alyssa 2

Temporal Parallelism

- Tray 1: Ben 1
- Tray 2: Ben 2
- Tray 3: Ben 3
Parallelism

• The goal of parallelism in general is to increase **throughput**, the amount of “work” done per unit time
  • In a factory it would be the number of products generated per unit time
  • For a processor, the number of instructions executed per clock cycle
• A computer could exploit both forms of parallelism simultaneously
  • Spatial parallelism: two or more CPUs (or CPU cores) execute separate instructions at the same time
  • Temporal parallelism: each CPU executes multiple instructions at once by executing different stages of the instructions simultaneously (this is pipelining)
Parallelism

• If an instruction (or step of an assembly/manufacturing process) can be divided into \( N \) steps that require the same amount of time, then we should be able to perform \( N \) instructions at once.

• In an ideal situation, this means that, for a fixed clock frequency, we can execute instructions \( N \) times faster than for the single-cycle CPU.

• In reality the pipelined CPU won’t be \( N \) times faster (due to overhead), but its throughput will be sufficiently higher to make the extra complexity of pipelining worth implementing.
Five Pipeline Stages

• In our multi-cycle implementation we have already split each MIPS instruction into five smaller tasks.
  • Instruction fetch (IF)
  • Instruction decode and register file read (ID)
  • Execution of arithmetic operation or address calculation (EX)
  • Data memory access (MEM)
  • Write back to register file (WB)

• By cascading the execution of each of these steps, we can increase the throughput of our datapath
Pipelined Execution

e tc.
Single-cycle vs. Pipelined CPU

- Timing diagrams: (a) single-cycle processor, (b) pipelined processor
Pipeline Control

• However, with this modification comes increased complexity
  • Extra control is required to manage the execution of multiple instructions simultaneously
  • Extra registers are required to hold the intermediate values of each instruction between stages
  • Contention for resources: what to do when two instructions need to use the same piece of hardware?
    • Example: PC+4 during Instruction Fetch at the same time as the ALU operation for another instruction.
    • Both instructions cannot use the same ALU simultaneously. We must add hardware.
Pipeline Control

• Consider also when the flow of the program changes (conditional branches/jumps)

• In these scenarios we will have begun execution of the next instruction(s) before knowing if the branch will be taken or not (which will be determined in the ALU stage)

• How do we stop their execution? We will see the solution to this problem a little later.
Pipeline Performance

• The maximum speed of the CPU will depend on how many stages we have
  • More stages means a higher clock speed because less work is performed per stage
• But, not all instructions will require all stages. In the multicycle CPU we would just skip unused stages.
• In the pipelined CPU we cannot eliminate these stages because other instructions are being executed (but are in different stages of the pipeline)
  • So we cannot simply “skip ahead” to the next instruction
• Not every stage may take the same amount of time, but the critical path will still dictate the length of the clock cycle and the length of execution for each instruction
Pros and Cons of Pipelining

• Pros:
  • Increased throughput
  • Parallelism

• Cons:
  • Individual instruction execution time may increase
  • Increased complexity in the datapath (its hardware and its control)
Pipeline Performance Comparison

• Assume time for instruction stages is:
  • 100 ps for register read or write
  • 200 ps for other stages

• For the single cycle datapath, the minimum cycle time is the length of the longest instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instr Fetch</th>
<th>Register Read</th>
<th>ALU Op</th>
<th>Memory Access</th>
<th>Register Write</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>100 ps</td>
<td>800 ps</td>
</tr>
<tr>
<td>sw</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td></td>
<td>700 ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td></td>
<td>100 ps</td>
<td>600 ps</td>
</tr>
<tr>
<td>beq</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td></td>
<td></td>
<td>500 ps</td>
</tr>
</tbody>
</table>
Pipeline Performance Comparison

• For the multicycle datapath, the minimum cycle time is the length of the longest stage (200 ps in this example), but each instruction requires multiple stages

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instr Fetch</th>
<th>Register Read</th>
<th>ALU Op</th>
<th>Memory Access</th>
<th>Register Write</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>1000ps</td>
</tr>
<tr>
<td>sw</td>
<td>200 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td></td>
<td>800ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td></td>
<td>200 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>beq</td>
<td>200 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td></td>
<td></td>
<td>600ps</td>
</tr>
</tbody>
</table>
Pipeline Performance Comparison

• For pipelining, we are executing in a multicycle datapath. Therefore the minimum cycle time is the length of the longest stage.

• However since we are executing multiple instructions simultaneously, we cannot skip stages
  • All instructions must “execute” all five stages in the same order
Pipeline Performance Comparison

Single-cycle ($T_c = 800\text{ps}$)

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Reg</th>
<th>ALU</th>
<th>Data access</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, 100\text{($0$)}$</td>
<td>800 ps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $2, 200\text{($0$)}$</td>
<td>800 ps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $3, 300\text{($0$)}$</td>
<td>800 ps</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kevin McDonnell
Stony Brook University – CSE 220
Pipeline Performance Comparison

Program execution order (in instructions):

- lw $1, 100(0)
- lw $2, 200(0)
- lw $3, 300(0)

Time:

200 ps

Instruction fetch:

Pipelined ($T_c = 200ps$)
Design with the End in Mind

• The MIPS ISA (instruction set architecture) was designed with pipelining in mind. To make pipelining easier:
  • All instructions have the same length (32 bits). This makes it easy to fetch and decode in one cycle.
  • It has few and regular instruction formats: can decode and read registers in one step
  • Load/store addressing: can calculate address in 3rd stage, access memory in 4th stage
  • Alignment of memory operands: memory access takes only one cycle because memory addresses are word-aligned. To read a word from memory requires only one lw instruction.
Pipelined Datapath Design

• As in the multicycle datapath, there are five stages, with one execution step per stage
  • IF: Instruction fetch from instruction memory (IM)
  • ID: Instruction decode & register read (RF)
  • EX: Execute operation or calculate address
  • MEM: Access memory operand from data memory (DM)
  • WB: Write result back to register file (RF)
• Reading and writing the memory and register file and using the ALU typically induce the biggest delays in the CPU
• These five pipeline stages were chosen so that each stage involves exactly one of these slow steps
Abstract View of the Pipeline

- Note how in each cycle, each of the (five) instructions being executed is in a different stage.
- A shaded stage indicates that the stage is being used.
- A white stage means that this stage is not being used.
Abstract View of the Pipeline

For the register file, *writes take place before reads*, which is the opposite from what the multicycle CPU does.

This ordering is depicted by the shading of the RF stages.
Pipelined Datapath Design

• Register writes happen before reads so that data can be written and read back within a single cycle
• Now, to support pipelining in the multi-cycle datapath, we would need to make some modifications:
  • Split the memory into separate memories, as was in the single-cycle datapath
  • Put the PC+4 adder and the Branch adder back in as they were in the single cycle datapath
  • Fundamentally, what are doing is to take the single-cycle datapath and add registers and other components as necessary to implement a pipelined datapath
• One major problem we will have to address during the design process is hazards
Pipelined Datapath Design

• A hazard occurs when the results of one instruction is needed by a subsequent instruction before the former instruction has completed.

• What would happen below if the add instruction read from $s2$ instead of $t2$?

![Diagram of pipeline stages](image)

lw $s2$, 40($0)
add $s3$, $t1$, $t2$

• We would have a **data hazard** because the **add** instruction in cycle 3 would not be able to read out $s2$ because it would not be written until cycle 5. Later on we will see different ways of addressing this kind of problem.
Single-cycle → Pipelined CPU

- Here’s the single-cycle datapath with some extra gaps between elements where we will insert pipeline registers to divide the datapath into five stages
Single-cycle $\rightarrow$ Pipelined CPU

• Since each stage of the pipeline will be executing a different instruction, all the data required for the next stage that comes from the current stage must be stored in registers.

• These registers are placed where the $Instr$, $A$, $B$, $Data$ and $ALUOut$ registers in the multicycle datapath resided and are expanded to hold more values.

• These registers are called **pipeline registers** and are sized only to hold the number of bits required to be stored.
Single-cycle $\rightarrow$ Pipelined CPU

- The letters F, D, E and M are appended to signals to indicate the stage in which they reside
Single-cycle → Pipelined CPU

• Consider the register file for a moment
• It’s drawn in the Decode (ID) stage because it is read during that stage (to provide operands to the ALU)
• However, it is written in a different stage, Writeback (WB)
  • Both the data and the write address come from Writeback
• This feedback introduces a hazard
  • Anytime you have data traveling from a later stage back to an earlier stage (i.e., to the left) we have a hazard
• So in fact our drawing has a mistake in it because the address of the register to write to (given as WriteRegE) comes from Writeback (WB), not Execute (EX)
Corrected Pipelined Datapath

- Now remember that the address of the register to write is given as part of the instruction.
- So what we need to do is to pipeline the WriteReg signal through the Memory and Writeback stages.
Corrected Pipelined Datapath

To see why this is necessary, consider this example from earlier.

In cycle 5 the `lw` instruction is supposed to write a word into register `$s2` (which is `rt`).

During cycle 5, however, `rt` is `$s5` because `sub` is in EX.

Had we not pipelined `WriteReg` to the Writeback stage the `lw` would write the word to `$s5` instead of `$s2`. 
Pipelined Processor Control

• Believe it or not, we can use the same control unit for the pipelined CPU as for the single-cycle CPU
• However, the control signals themselves must be pipelined along with the data so that they remain synchronized with the instruction
• This means we must also pipeline the $RegWrite$ control signal to enable writing at the proper time
  • $WriteReg$ identifies which register we are writing to
  • $RegWrite$ is a control signal that indicates if we are writing a word to the register file
Pipelined Processor with Control
Instruction Execution: \( \text{l}w \)

- For a given instruction, only those components within a stage are used for that instruction.
- During Instruction Fetch (IF), PC+4 is computed and forwarded to the next stage because it may be needed to calculate a branch address.
Instruction Execution: $\text{l}w$

- Instruction Decode (ID): store $\text{Reg}[rs], \text{Reg}[rt]$ and the sign-extended immediate in the pipeline registers
- Note: branch address is not computed in this stage
Instruction Execution: $lw$

- Execute (EX): store $\text{Reg[rs]} + \text{SignImm}$ and $\text{Reg[rt]}$ in the pipeline registers
- All values are read from pipeline registers
Instruction Execution: \( \text{lw} \)

- Memory Access (MEM): read \text{DataMem}[\text{ALUOutM}]\) and store it in pipeline register
Instruction Execution: \( lw \)

- Writeback (WB): write \textit{ReadDataW} to \textit{Reg[rt]}
Data Hazards

• The example below shows a read after write (RAW) hazard
• This occurs when an instruction writes a register and subsequent instructions read that register
• Some of the later instructions will read the wrong value

add $s0, $s2, $s3
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
Data Hazards

• A data hazard is not the only kind of hazard

• A control hazard occurs when the decision of what instruction to fetch has not been made by the time the fetch takes place

• A structural hazard occurs when two instructions want to access the same resource, such as memory or an ALU
  • This kind of hazard is avoided by adding extra hardware, such as adders and memories

• We will add a hazard detection unit to the pipelined processor that detects and handles hazards appropriately so that the processor executes the program correctly
Forwarding

• Some data hazards can be solved by **forwarding** a result from the MEM or WB stage to a dependent instruction in the EX stage

• Consider the example from earlier. Note that the sum from the **add** instruction is computed in cycle 3 and is needed by the **and** instruction in cycle 4.

• We should be able to forward the result from the first instruction to the next to resolve the RAW hazard (red)

```
add $s0, $s2, $s3
and $t0, $s0, $s1
```
Forwarding

- In fact, we can do several data forwards to solve many of the RAW hazards from the example

1 2 3 4 5 6 7 8

Time (cycles)

add $s0, $s2, $s3

and $t0, $s0, $s1

or $t1, $s4, $s0

sub $t2, $s0, $s5

- This means we need to change the sources of inputs (e.g., operands to the ALU, words to be written to the register file) from what would normally be used as inputs
Implementing Forwarding

• Multiplexers come to the rescue!
• To change the source of inputs to the stages (i.e., so we can forward results), we add muxes in front of the ALU to select inputs from:
  • the ID/EX pipeline registers (normal operation), or
  • the MEM stage (forwarding being used), or
  • the WB stage (forwarding being used)
• We forward from MEM to EX or WB to EX when we need the result of the ALU
• We might also forward from WB to EX when we are doing a `lw` and need the result sooner
Implementing Forwarding

• To implement this logic, the hazard detection unit receives the two source registers (from the EX stage) and the destination register (from the MEM and WB stages)

• It also takes $\text{RegWrite}$ from the MEM and WB stages to see if the destination register will actually be written

• The hazard detection unit should forward from a stage if that stage will write a destination register and that destination register matches a source register
  • $0$ is hardwired, though, and is never forwarded

• If both MEM and WB have matching destination registers, then the MEM stage should take priority because its instruction is the more recently started instruction
Implementing Forwarding
Implementing Forwarding

- Input 00 corresponds with normal operation, so the source operand is taken from the register file
- 01 corresponds with a forward from WB
- 10 corresponds with a forward from MEM
Implementing Forwarding

• Here is how the hazard detection unit determines the control signal for the SrcA mux (SrcB is similar)

if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
  ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW)) then
  ForwardAE = 01
else
  ForwardAE = 00
When Forwarding Fails

- Time travel is not possible! We cannot forward the output of one stage to the beginning of another stage during the same (or a previous) clock cycle.
- In other words, not all data hazards can be avoided.
When Forwarding Fails

• This example shows that \texttt{lw} has a \textbf{two-cycle latency} because the loaded word is not available for dependent instructions until two cycles after the \texttt{lw} receives the word from memory.

\texttt{lw \$s0, 40(\$0)}

\texttt{and \$t0, \$s0, \$s1}

\texttt{or \$t1, \$s4, \$s0}

\texttt{sub \$t2, \$s0, \$s5}
Stalling the Pipeline

- In some cases the only option is to **stall** the pipeline, which means we hold up an operation until the data is available.
- See the example below: the **and** and **or** instructions must be stalled one cycle to wait for the **lw** to get the word.
- **and** stalls in the ID stage for one cycle; **or** stalls in the IF stage for one cycle.

```
lw $s0, 40($0)
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
```
Stalling the Pipeline

- Note that the EX stage is unused in cycle 4, MEM is unused in cycle 5, and WB is unused in cycle 6
- This unused cycle propagating through the pipeline is called a **bubble**
- It is implemented as a **nop** instruction \((\text{sl}1 \; $0, \; $0, \; 0)\), which is encoded as 32 0s in the instruction

```
lw \$s0, 40(\$0)
and \$t0, \$s0, \$s1
or \$t1, \$s4, \$s0
sub \$t2, \$s0, \$s5
```
Stalling the Pipeline

- The bubble is created by zeroing out the control signals for the EX stage so no changes to the pipeline registers occur.
- All the instructions that are in earlier stages (e.g., IF and ID) need to be stalled also.
- Note how the ID stage of the or is stalled and the beginning of execution of the sub is stalled.

lw $s0, 40($0)
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
Modified Pipeline for \( \text{lw} \) Stalls

- If an instruction in the EX stage is a \( \text{lw} \) and its destination register (\( r_tE \)) matches either source operand of the instruction in the ID stage (\( r_sD \) or \( r_tD \)), that instruction must be stalled until the source operand is ready.
- The hazard detection unit will force the IF and ID stage pipeline registers to hold their old values by asserting new \( \text{StallF} \) and \( \text{StallD} \) control signals.
- A new control signal called \( \text{FlushE} \) is asserted to clear the contents of the EX pipeline register and introduce the bubble.
- Pseudocode:
  
  \[
  \text{lws\text{\text{stall}}} = ((r_sD==r_tE) \text{ OR } (r_tD==r_tE)) \text{ and MemtoRegE} \\
  \text{StallF} = \text{StallD} = \text{FlushE} = \text{lws\text{\text{stall}}}
  \]
Modified Pipeline for lw Stalls
Control Hazards

• The `beq` instruction causes a **control hazard**.
• The pipelined processor cannot determine which instruction to fetch next because the branch decision is not made until cycle 4 of the instruction
• We could simply stall the pipeline three cycles for every branch, but that would degrade system performance
• Another option: **branch prediction**
• Suppose we always assume that branches are not taken
• When our prediction is correct, nothing goes wrong
• But if the branch *is* taken, this means we have fetched three additional instructions incorrectly
• These instructions must be **flushed** by clearing the pipeline registers for those instruction
Flushing the Pipeline

- In this example, instructions at addresses 0x24, 0x28 and 0x2C were fetched unnecessarily.
- We flush them and start executing the instruction at the branch target address (0x64).
Branch Prediction

- If we could somehow make the branching decision during the ID stage, then we would lose only one cycle.
- Solution: use a dedicated equality comparator in the ID stage to compare the operands that are read from the register file.
- If the branch is taken, then only one instruction must be flushed and the pipeline is stalled for one cycle.
Branch Prediction

20  beq $t1, $t2, 40
24  and $t0, $s0, $s1
28  or $t1, $s4, $s0
2C  sub $t2, $s0, $s5
30  ...
...
64  slt $t3, $s2, $s3
Control Hazard Handling
Control Hazard Handling

• This approach introduces a RAW hazard – what if rs or rt was computed by a previous instruction and hasn’t been written to the register file yet?

• We need to forward the result we need (if it’s available) or stall the pipeline

• If the result is in the WB stage, there’s no hazard – the result will be written at the start of the cycle and read at the end

• If the result is in the MEM stage, then we can forward it to the ID stage

• If the result is in the EX stage or the result is coming from a lw instruction that’s currently in the MEM stage, the pipeline is stalled one cycle
Control Hazard Handling
Control Hazard Handling

- The forwarding logic used in the ID stage is:
  \[
  \text{ForwardAD} = ((rsD \neq 0) \land (rsD == WriteRegM) \land \text{RegWriteM}) \\
  \text{ForwardBD} = ((rtD \neq 0) \land (rsD == WriteRegM) \land \text{RegWriteM})
  \]

- The stall detection logic handles both an ALU instruction in the EX stage and a `lw` instruction in the MEM stage:
  \[
  \text{branchstall} = (\text{BranchD} \land \text{RegWriteE} \land \\
  (\text{WriteRegE} == rsD \lor \text{WriteRegE} == rtD)) \lor \\
  (\text{BranchD} \land \text{MemtoRegM} \land \\
  (\text{WriteRegM} == rsD \lor \text{WriteRegM} == rtD))
  \]

  \[
  \text{StallF} = \text{StallD} = \text{FlushE} = \text{lwstall or branchstall}
  \]
Pipelined CPU with Hazard Handling
Instruction Reordering

- Yet another way to avoid stalls is to reorder instructions to eliminate stalls, but still have the program produce the correct result.
- Suppose we have the following code that implements this algorithm: $A = B + E; \ C = B + F;
  \begin{align*}
  &\text{lw } \$t1, \ 0(\$t0) \\
  &\text{lw } \$t2, \ 4(\$t0) \\
  &\text{add } \$t3, \ $t1, \ $t2 \\
  &\text{sw } \$t3, \ 12(\$t0) \\
  &\text{lw } \$t4, \ 8(\$t0) \\
  &\text{add } \$t5, \ $t1, \ $t4 \\
  &\text{sw } \$t5, \ 16(\$t0)
  \end{align*}
- Because of the highlighted data dependencies (hazards), this code will take 13 cycles to execute. Why?
Instruction Reordering

• Reordering the instructions as shown on the right will eliminate the data hazards. The reordered code will take 11 cycles to execute.

```
lw  $t1, 0($t0)    lw  $t1, 0($t0)
lw  $t2, 4($t0)    lw  $t2, 4($t0)
add $t3, $t1, $t2  add $t4, 8($t0)
sw  $t3, 12($t0)   add $t3, $t1, $t2
lw  $t4, 8($t0)    sw  $t3, 12($t0)
add $t5, $t1, $t4  add $t5, $t1, $t4
sw  $t5, 16($t0)   sw  $t5, 16($t0)
```
Example #1: Hazards

- Suppose we have the following MIPS code. Circle registers to indicate dependencies between instructions that cause RAW hazards.

  ```
  add $s0, $t1, $t2
  sub $s0, $s0, $t1
  nor $t3, $s0, $s0
  nor $t4, $t6, $t7
  and $t3, $s0, $s0
  ```
Example #2: Hazards

- Assuming no forwarding hardware is available, draw bubbles to show how the pipeline would be stalled to correctly execute the code.

```
addd $s0, $t1, $t2

sub $s0, $s0, $t1
nor $t3, $s0, $s0
nor $t4, $t6, $t7
and $t3, $s0, $s0

add $s0, $t1, $t2
sub $s0, $s0, $t1
nor $t3, $s0, $s0
nor $t4, $t6, $t7
and $t3, $s0, $s0
```
Example #3: Hazards

• Assuming forwarding hardware is available, show the forwarding paths and stalls needed to execute the same instructions

```
add $s0, $t1, $t2
sub $s0, $s0, $t1
nor $t3, $s0, $s0
nor $t4, $t6, $t7
and $t3, $s0, $s0
```
Example #4: Hazards

- Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions.

```plaintext
add $s5, $s2, $s7
sub $s4, $s5, $s1
or $s3, $s4, $s5
nor $s5, $s3, $s3
```
Example #5: Hazards

- Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions.

```
lw $s1, 8($t2)
sub $s2, $s1, $t1
add $t2, $s1, $s2
sw $s1, 8($t2)
or $t3, $s1, $t4
```
Example #6: Hazards

- Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions.

\[
\begin{align*}
\text{sw } & \text{ } \text{s5, 20(s3)} \\
\text{lw } & \text{ } \text{s5, 12(s2)} \\
\text{ori } & \text{ } \text{s2, s5, 0xFF} \\
\text{add } & \text{ } \text{s6, s5, s2} \\
\text{lw } & \text{ } \text{s4, 0(s6)} \\
\text{lw } & \text{ } \text{s7, 0(s4)} \\
\text{sub } & \text{ } \text{s4, s6, s3}
\end{align*}
\]
Example #7: Hazards

- Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions. Assume the branch is taken and there is no comparator.

```assembly
lw $s1, 12($s2)  
beq $s2, $s5, skip  
add $s6, $s5, $s2  
add $s4, 0($s7)  
add $t1, $t2, $t3  

lw $s1, 12($s2)  
beq $s2, $s5, skip  
add $s6, $s5, $s2  
add $s4, 0($s7)  
add $t1, $t2, $t3  

lw $s4, 0($s7)  
add $t1, $t2, $t3  

lw $s4, 0($s7)  
add $t1, $t2, $t3  

xor $s7, $s5, $t8  
sub $s4, $s4, $s2  

xor $s7, $s5, $t8  
sub $s4, $s4, $s2
```
Example #8: Hazards

• Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions.

```
lw $s5, 12($s2)
add $s2, $s4, $s5
dl $s5, 12($s2)
add $s2, $s4, $s5
dl $s2, 0($s4)
add $s2, $s4, $s4
add $s3, $s2, $s6
```

The blue line here shows the WriteData line in the datapath.

We could instead forward $s2 using the green path.
Example #9: Hazards

- Using the abstract pipelined datapath figure, show the forwarding paths and stalls needed to execute the following instructions.

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

The blue line here shows the WriteData line in the datapath.

We could instead forward $t3 using the green path.