CSE220 - MIPS Pipelining

- What we have seen so far 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.
- One disadvantage, is that the datapath has several parts/components, which sit idle when other parts are in use.
  - In the multicycle machine we tried to optimize this some by combining units (single ALU, one memory for data and instructions). We also try to do as much work as possible in each state.
- The goal of Pipelining is to increase this utilization by handling several instructions at a time.

**Ex: Laundry**
- When we do laundry there are 4 steps: Load Washer, Move to Dryer, Fold Laundry, Store Laundry (put Away)
- Assume we have 4 loads of laundry to do.
- If we do laundry the same way we have been processing instructions, one at a time, it would take use 16 time units to do all the laundry form start to finish.
  - If we however, do laundry more efficiently: we use the washer again as soon as the previous load is done. We can finish much faster.
  - Since each load is at a different stage of the laundry process, we can do multiple loads at a time. The total time is now 50% of the original.
  - Once the ‘pipeline’ is full (when the fourth load – Task D – starts), we can do the laundry four times as fast.

**Pipelined Datapath Execution**
- In our multi-cycle implementation we have already split each MIPS instruction into 5 smaller tasks.
- Instruction fetch (IF)
- Instruction decode & register file read (ID)
- Execution of arithmetic operation or address calculation (EX)
- Data memory access (MEM)
- Write back to register (WB)

- By cascading the execution of each of these cycles, we can increase the throughput of our datapath.

- 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.
  - What happens when two instructions need to use the same piece of hardware…. We must add additional hardware to the datapath to remove these situations
    - Ex: PC+4 during Fetch at the same time as the ALU operation for another instruction. Both cannot use the same ALU
  - 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 (ALU stage). How do we stop their execution?

- The maximum speed of the CPU will depend on how many stages we have.
  - More stages → high clock speed. Why? Less work to perform per stage.

- But, not all instructions will require all stages……
  - Can I eliminate these stages? No. because other instructions require the hardware during these stages, I cannot just skip ahead to the next stages/instruction.

- Furthermore, each stage may not take exactly the same amount time.
  - But the critical path will still dictate the length of the clock cycle and the length of execution for each instruction (all stages).

**Pros/Cons of Pipelining**
- **Pros:**
  - Increased throughput
  - Parallelism
- **Cons:**
  - Individual instruction execution time may increase
- Increased complexity in the datapath (hardware/control)

**MIPS Pipelining Datapath**

- Pipelining increases the performance of the machines, but at the cost of greater complexity to the datapath.

**Ex: Performance Comparison**
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- For Single cycle datapath:
  - The minimum cycle time is the length of the longest instruction

<table>
<thead>
<tr>
<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>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>100ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

- For Multi cycle datapath:
  - The minimum cycle time is the length of the longest stage, but each instruction requires multiple stages

<table>
<thead>
<tr>
<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>200ps</td>
<td>200ps</td>
<td>200ps</td>
<td>200ps</td>
<td>1000ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>200ps</td>
<td>200ps</td>
<td>200ps</td>
<td>800ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>200ps</td>
<td>200ps</td>
<td>200ps</td>
<td>800ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>600ps</td>
</tr>
</tbody>
</table>

- For pipelining, we are executing in a multi-cycle 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 5 stages in the same order.
The MIS ISA was designed with Pipelining in mind. To make pipelining easier:

- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
- 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

As in the multicycle datapath, there are five stages, one execution step per stage:

- IF: Instruction fetch from memory
- ID: Instruction decode & register read
- EX: Execute operation or calculate address
- MEM: Access memory operand
- WB: Write result back to register
Abstract View of 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.
  - 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

- Register writes happen before reads so that data can be written and read back within a single cycle
- In order to support pipelining in the multi-cycle datapath, modifications need to be made.
  - Split the memory into separate memories as was in the single cycle datapath
  - Put the PC+4 adder and the Branch adder back as was in the single cycle datapath
- Additionally, each component of the datapath can only be used by a single instruction per clock cycle. If two instructions need to access/store data to the same component, there is a conflict which is called a **HAZARD**. Other hazards are the result of branch or jump instructions which modify the program counter and change the standard flow of instruction execution. (Hazards will be discussed later)
- Each stage of the pipeline will be executing a different instruction, therefore all the data required from the current stage to the next stage must be stored in registers. These registers are placed where the IR, 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 (not necessarily multiples of 32).

- The registers are named based on the stages they sit between. For example, between the Instruction Memory and the Register File is the IF/ID.
- Specifically, how big are these registers (data only)?
  - IF/ID register: 64 bits (Instruction & PC+4)
  - ID/EX register: 138 bits (PC+4, Reg Read 1, Reg Read 2, Sign Extended I value, Rt, Rd)
  - EX/MEM: 102 bits (Branch address, ALUOut, Zero bit, Reg Read 2, WriteRegDestination)
  - MEM/WB: 69 bits (ALUOut, Data, WriteRegDestination)

Note, ID/EX, EX/MEM, MEM/WB will be expanded later to hold more values (for example, the control signals).
Instruction Execution in Pipeline Datapath

- Consider the execution of a LW instruction through different stages of the pipeline
- Each stage takes the input values from the pipeline registers. Note that only the components within that particular stage are used for the instruction.
- Instruction Fetch (IF)
  - Note that the PC+4 value is calculated, and it is forwarded to the next stage. Why? Because the PC+4 value is needed to calculate the Branch/Jump address in later stages.
    - # PC <= PC + 4
    - # IF/ID <= Instruction Mem[PC]
  - Note that the PC+4 value is calculated, and it is forwarded to the next stage. Why? Because the PC+4 value is needed to calculate the Branch/Jump address in later stages.

- Instruction Decode (ID)
  - Note how the branch address is not calculated in the decode cycle.
    - # ID/EX <= Reg[rs]
    - # ID/EX <= Reg[rt]
    - # ID/EX <= sign extended IF/ID (immediate value)
- **Execute (EX)**
  - # EX/MEM <= Reg[rs] + SignExtended Immediate
  - # EX/MEM <= Reg[rt]
  - All read from ID/EX pipeline registers
- **Memory (MEM)**
  - # MEMWB <= Data MEM[EX/MEM ALUOut], both read from EX/MEM pipeline registers

- **Write Back (WB)**
  - #Reg[rt] <= Mem/WB Data value
  - Notice that the register file Write Register input is incorrectly specified. The register number must be from the current instruction executing and not from the instruction currently in the Decode stage. Therefore, we must modify the datapath to forward the write register from the LW instruction though all the stages in order to have the information available at the time of the WB. (Note, that we do not forward the entire instruction and there is no Instruction register to hold the instruction during the entire execution.)
• Consider the SW instruction
  o The fetch (IF) and decode (ID) stages never change, as they are instruction in-dependent.
  o The execute stage (EX) still calculates the memory address
  o The MEM stage, writes back the data to memory from the Reg[rt]
  o The WB stage is not needed for the SW instruction. No additional writing to registers is needed. Even though the instruction does not need to perform this stage, no other instruction can use this hardware/time. Why?

**Pipelined Datapath Control**

• Below is the control signals which are required in each stage in the simplified view of the datapath

  ![Diagram of control signals](image)

• The control of the multi-cycle datapath was calculated by a single control unit in the decode stage of execution. The microcontroller/finite state machine specified the control signals for each instruction for each stage of execution.

• It is inefficient to have a finite state machine for each instruction in each stage of the pipeline. Especially, when we don’t even save the full instruction after the Decode stage.

• In the single cycle datapath all control signals were calculated for the instruction in the decode stage. The pipelined datapath reverts to this approach. Note that since the instruction itself is not saved after the decode stage, the control signals for each cycle cannot be calculated during each cycle, instead they are all calculated during the decode stage and passed to the following cycles.

**MIPS Pipelining Hazards**

• Hazards are situations, caused by the sequence of MIPS instructions or the hardware, that prevent or complicate the execution of the next instruction in the pipeline.

• There are 3 types of hazards:
  o *Structural hazards:* when two instructions want to access the same resource, aka a required resource is busy
  o *Data hazards:* required data is needed from instructions in the pipeline, aka need to wait for previous instructions to complete its data read/write
Control hazards: the next instruction to enter the pipeline is not known, aka deciding the control depends on the previous instruction

**Structural Hazards**
- These hazards occur when there is a conflict for use of a resource.
- An excellent example is the memory in the multicycle datapath.
- **Ex:** In the multicycle datapath the instruction and data memory were combined this reduced the amount of required hardware. Load and store instructions required access to the data in this memory during the MEM stage. Also the same memory is accessed during the Fetch stage to obtain the instruction to execute.
  - Fetching the instruction in the pipeline while in the MEM stage of a lw/sw instruction is executing would not be possible. Therefore the Fetch would need to be delayed by a clock cycle, aka *STALL*.
- Pipelined datapaths require separate instructions and data memories to eliminate this structural hazard.

**Data Hazards**
- A data hazard occurs when an instruction depends on the completion of a previous instruction. IE. Next instruction cannot finish execution because data needed is not available yet.

  **EX:**
  ```
  add $s0, $s2, $s3
  and $t0, $s0, $s1
  ```
  - Consider the above instructions in the pipeline one after another.

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

  - Add instruction: The result of the $s2+$s3 operation is not stored back into $s0 until the WB stage (5th cycle of execution)
  - Sub instruction: The value of $s0 is read from the register file during the ID stage (second stage).
  - The subtract instruction will fetch the incorrect value of $s0 because the fetch is performed while the add instruction is in the EX stage.
  - As a result of this issue, to execute these two instructions correctly, we would need to stall the execution of the sub instruction by 2 clock cycles. In this way the ID (register read) would occur after the WB (register write) of the add instruction.

- Stalling the pipeline negatively impacts performance. Therefore, in order to resolve data hazards additional hardware is added to the datapath. There are multiple approaches to try to eliminate stalls.

  **Forwarding/Bypassing**
  - Instead of waiting until the write back occurs, we will “forward” the data as soon as it is available
  - We do not wait until the value is not stored back in the register
  - However, we need new connections (wires and multiplexor options) in the datapath
Forwarding does resolve data hazards in many cases, but not all.
- Data can never be sent backwards in time!
- Ex: several data forwards to solve many of the read-after-write (RAW) hazards

- Ex: $lw \ $s0, 20($t1)
  $sub \ $t2, \ $s0, \ $t3$

- The $s0$ value from the load word instruction is not available until after the MEM stage
- This value is required by the sub instruction after ID but before EX stage.
- In this case, forwarding and stalling is needed.
- When an R-format instruction following a load tries to use the data
- Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible.
- Figure is actually a simplification!
- Don’t know until after Decode of Sub, that a stall is necessary……
- The value to be placed into $s0$ is not available until after being read from memory in the 4th stage. However it is still needed in the ID stage of subtract. A stall is required in this case.

- **Code Scheduling to avoid Stalls**
  - The order in which instructions are programmed can often be modified while still producing the same result.
  - One approach to eliminate stalls due to load instructions is to reorder the code to avoid the use of the load result in the next instruction.
  - Ex: $A = B + E; \ C = B + F;$
The left code sequence requires 2 stalls because the load instructions are immediately followed by references. The right code sequence, however, reorders the instructions to have at least 1 instruction between each load and its reference. The code produces the same result and the stalls are eliminated.

**Data Hazards & the Datapath**

- Consider the sequence:
  
  ```
  sub $2, $1, $3
  and $12, $2, $5
  or $13, $6, $2
  add $14, $2, $2
  sw $15, 100($2)
  ```

- Which instructions will potentially have issues? What type?
  - and, or (data hazard)
  - add, sw (data hazard)

- **How do we detect when to forward for these types of instructions?**
  - Register $2 is used by every instruction. The first instruction modifies the contents, and all remaining instructions use the new value.
  - The figure highlights the ID stage where $2 is read for all instructions following the first.

- Can all hazards be resolved using forwarding?
  - When is the correct value of the sub instruction calculated and available? After EX
  - The value from sub EX can be forwarded to the EX stage of the and instruction.
  - It can also be forwarded from the sub MEM to the EX stage of the or instruction.
  - With the add instruction because writes to the register file occur prior to the reads, there is no data hazard. The sw instruction has no data hazard as the sub instruction is complete prior to the ID stage of sw.
  - NOTE: General rule of thumb: All data hazards drawn backward in time need forwarding.

- How do we detect when forwarding is needed?
  - If the rs and rt registers of the current instruction are the rd register of the instruction before or the instruction 2 before, then forwarding is required.
  - Logically:
    - Forwarding from EX/MEM pipeline register (instruction before)
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
   ▪ Forwarding from MEM/WB pipeline register (instruction 2 before)
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
   • But only if the these 2 prior instructions are going to write to the rd register
     ▪ How do we know? Control signal RegWrite will be 1
     ▪ What if the rd register is $0$? Then there is no need to forward, because $0$ is always $0$.

• To enable forwarding modification to the datapath and pipelining registers are required. A Forwarding control unit is added.
  • In order to detect the need for forwarding we need to compare the rs and rt register number in the ID/EX stage of the current instruction with the rd register number of the 2 prior instructions.
  • Remember the rd register for each instruction is already passed through the pipeline for the WB stage.
  • The below figure modifies the datapath to introduce new multiplexors into the EX stage. These select between the rs and rt data coming from the 3 different places: the ID/EX (register file), the EX/MEM (ALUOut), the Mem/WB (Data/ALUOut delayed by 1 cycle)
  • In addition, we must consider the special case of an instruction trying to write to the $0$ register. Eg. add $0$, $t0$, $t2$
    ▪ In this case, the instruction is equivalent to a No Operation (nop) instruction. But assembly has no special constructs to prevent a programmer from writing this code, as the $0$ is a special case register.
    ▪ We need to set the control logic to NOT forward the data when the destination register (rd) is $0$. Therefore, we need to add the condition EX/MEM.RegisterRd ≠ $0$ and MEM/WB.RegisterRd ≠ $0$

  • The forwarding unit control these multiplexors through digital logic comparing the above mentioned conditions.
Logically when are the forwarding values used? For an EX hazard

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

The full datapath with hazard & forwarding unit

How do we detect when to stall?
- Stalls are required whenever the data required is not available at the time it is needed.
- This occurs when an instruction needs a value from a lw instruction
How do we detect when a stall is needed?
  o Check when current instruction is decoded in ID stage (rs and rt) is same as rd and the previous instruction is a lw instruction (the MemRead control signal will be 1)
  o Logically:
    - ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt))
      If detected, stall and insert bubble

How do we stall/insert a bubble in the pipeline?
  o Force the control values in the ID stage to 0. This means that the EX, MEM, and WB stages will do nothing (nop instruction)
  o Then we need to repeat the instruction again. To do this we prevent the PC update and update of the IF/ID register
    - The current instruction will be decoded again
    - The next instruction will be fetched again
    - 1-cycle stall allows for MEM to read data for lw and then it can be forwarded from the MEM/WB pipeline register to the EX stage.

Overall, stalls reduce the performance of the machine, but are required in order to produce the correct results.
Another approach is to have the compiler rearrange the code to avoid hazards and stalls.
This requires the compiler to have knowledge about the pipeline structure within the architecture it is compiling for.

**Control Hazards**

- Typical execution of instructions is one after another in memory. However, branch and jump instructions allow for the linear execution to be modified.
- During pipelining, the branch instruction will determine the next instruction to fetch. But the branch determination is not complete until the 4th stage MEM (The ALU zero AND Branch control is performed in MEM). The pipeline can’t and won’t always fetch the correct instruction. (correct if branch not taken, incorrect if branch taken).
- If the branch outcome is determined in the MEM cycle then 3 instructions have already started to be processed.
We can reduce this branch delay by moving the branch address calculation and perform the branch comparison to the ID stage. Additional hardware is required.

By performing the branch in the check in the decode stage, then when the branch is taken the control signals for the next instruction (in fetch stage) can be set to nop (IF.flush control signal).
What about when Data Hazards occurs with Branches?

- If one of the branch comparison registers is the rt register of the previous 2\textsuperscript{nd} or 3\textsuperscript{rd} ALU instruction there is a data hazard. We can resolve these issues with forwarding.

```
add $1, $2, $3  \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

```
add $4, $5, $6  \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

... 

```
beq $1, $4, target  \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

- If one of the branch comparison registers is the rt register of the previous ALU instruction or previous 2\textsuperscript{nd} load instruction, then the branch instruction must be stalled. There is no way to forward the data in time to complete the branch in ID.

```
lw $1, addr \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

```
add $4, $5, $6  \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

```
beq stalled  \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

```
beq $1, $4, target  \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

- If one of the branch comparison registers is the rt register of the previous load instruction, then the branch instruction must be stalled for 2 cycles. There is no way to forward the data in time to complete the branch in ID.

```
lw $1, addr \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

```
beq stalled  \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

```
beq stalled  \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

```
beq $1, $0, target  \hspace{0.5cm} \begin{array}{|c|c|c|c|c|}
\hline
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
\end{array}
```

In the MIPS pipeline the branch delay is not serious. However in longer pipelines and in superscalar pipelines, the branch penalty is more significant. In these cases, branch prediction is used, specifically dynamic branch prediction.
• The forwarding logic used in the ID stage is:
  \[\text{ForwardAD} = ((rsD \neq 0) \land (rsD = \text{WriteRegM}) \land \text{RegWriteM})\]
  \[\text{ForwardBD} = ((rtD \neq 0) \land (rsD = \text{WriteRegM}) \land \text{RegWriteM})\]

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

**Branch Prediction**
- Branch Prediction is another approach. In this case, a stall is only required if the prediction is wrong.
- Assume we predict that all branches are not taken, therefore you always fetch the next instructions after each branch.
What happens when you predict wrong?
- The instruction that was fetched has to be FLUSHED from the pipeline (turned into a No Operation, NOP instruction).

More realistic branch prediction methods exist, such as static branch and dynamic branch prediction.
- Static branch prediction is based on the typical behavior – loop or if statements
  - All backward branches are predicted to be taken
  - All forward branches are predicted not taken
- Dynamic branch prediction uses hardware to measure the actual branch behavior during execution.
  - Assume that the future behavior is the same as observed. When wrong, the pipeline is stalled and the correct instruction fetched. The history is then updated.