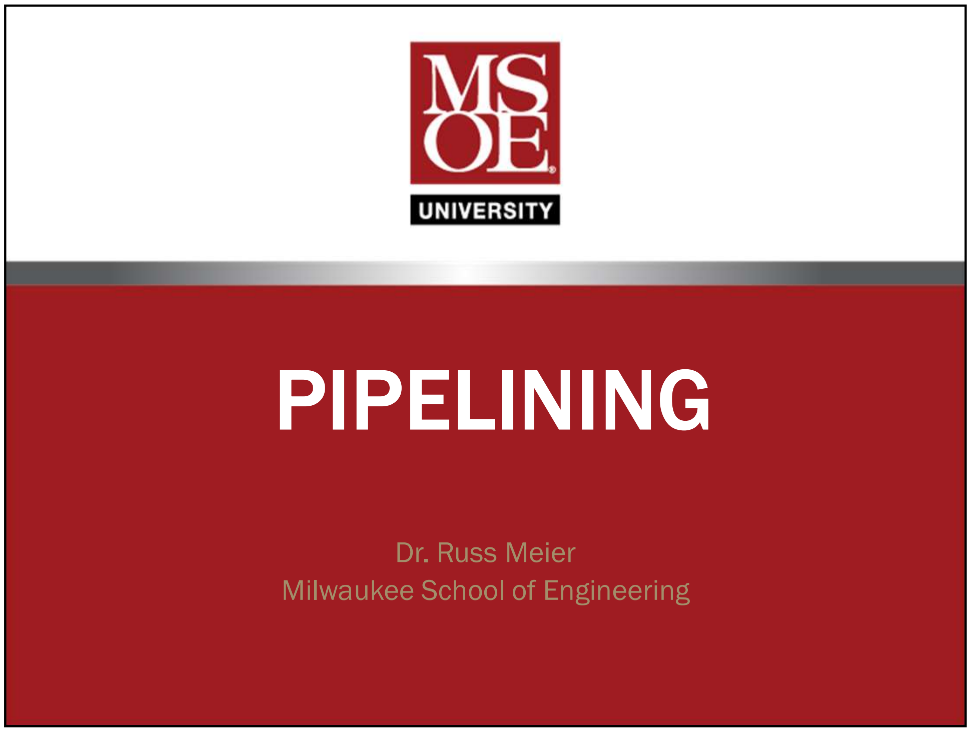
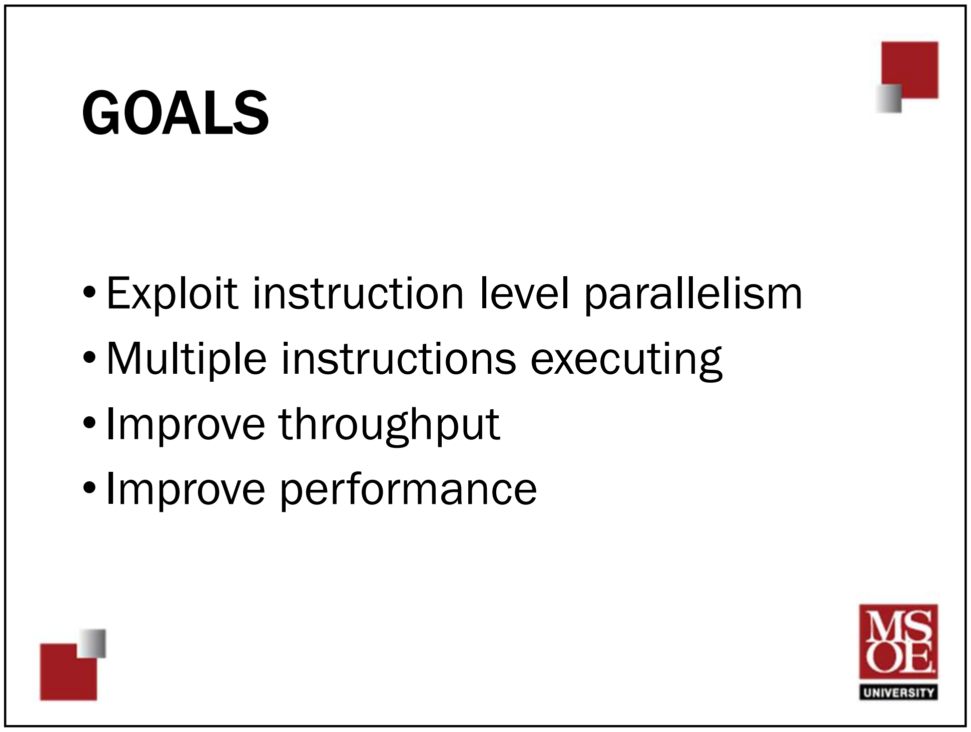


PIPELINING

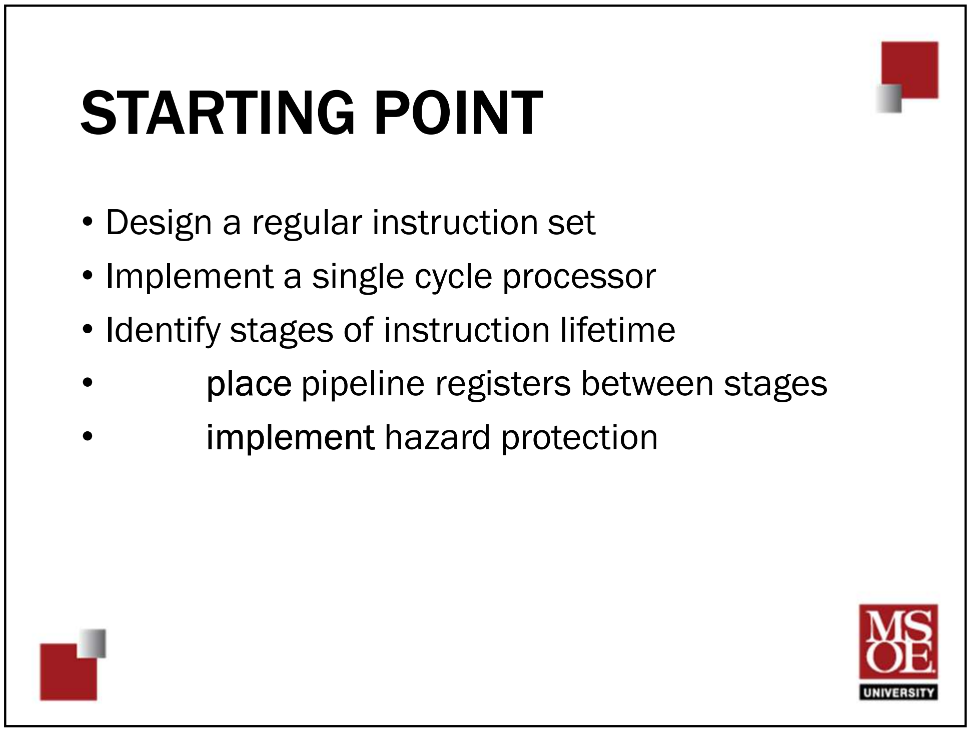
Dr. Russ Meier
Milwaukee School of Engineering



GOALS

- Exploit instruction level parallelism
- Multiple instructions executing
- Improve throughput
- Improve performance

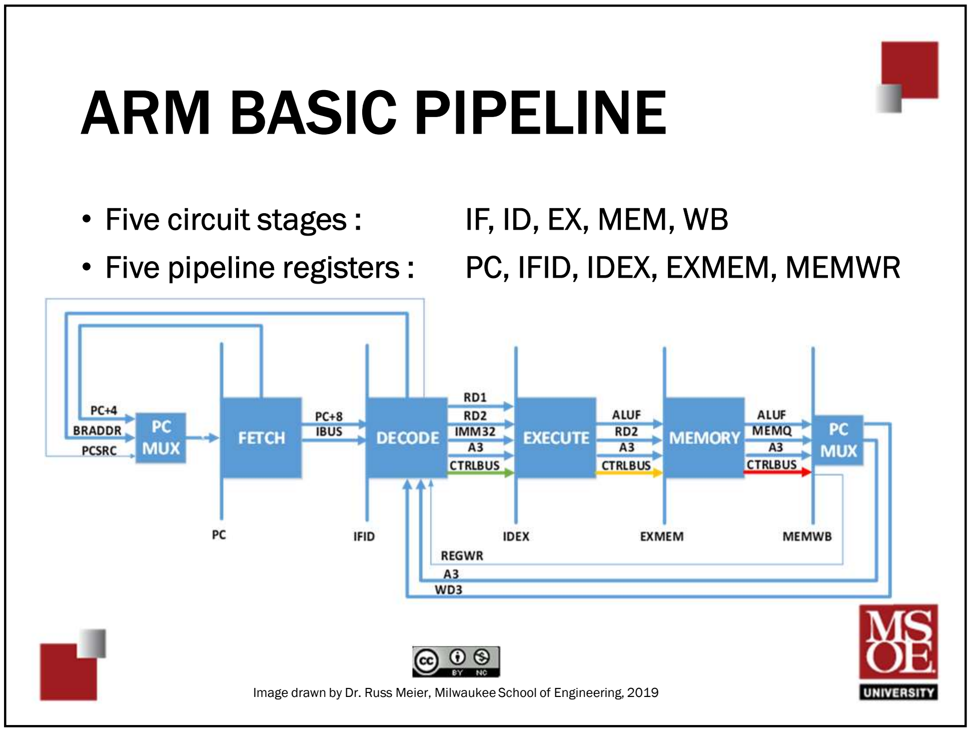
The goal of pipelines is to exploit instruction level parallelism. Parallel execution implies that multiple instructions are in some stage of execution at the same time. If we can efficiently use the processor circuits by filling them with some instruction at all times, then we should be able to improve throughput (the number of instructions completing per unit time) and performance (the total execution time of the program) when compared against the single-cycle processor where only one instruction is in the circuitry at any time.



STARTING POINT

- Design a regular instruction set
- Implement a single cycle processor
- Identify stages of instruction lifetime
- place pipeline registers between stages
- implement hazard protection

Pipelining requires a very regular instruction set. Regularity in the instruction set will lead to simpler circuitry. Simpler implies faster because there will not be as many gate delays. Often, a pipeline architect will begin by sketching a single-cycle processor. This allows the architect to identify the circuitry required to implement the instruction set and organize it into circuit **stages**. These stages of instruction lifetime are then separated by pipeline registers that sample data from one stage and advance it to the next. The architect must then consider the hazards that naturally exist in programming. These hazards are discussed later in the presentation. Hazards prevent the pipeline from remaining fully active at all times. In other words, certain hazards may ***stall*** the pipeline – allowing some instructions to move forward through later circuit stages while others remain stalled in the earlier stages.



ARM BASIC PIPELINE

- Five circuit stages : IF, ID, EX, MEM, WB
- Five pipeline registers : PC, IFID, IDEX, EXMEM, MEMWR

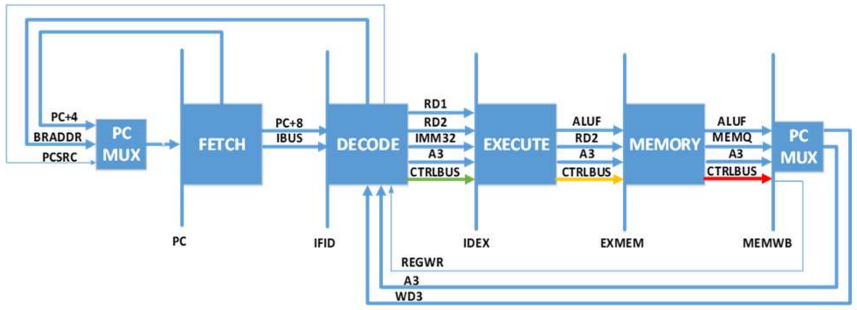


Image drawn by Dr. Russ Meier, Milwaukee School of Engineering, 2019

The ARM architects created a regular instruction set that can be easily pipelined. The diagram shows the signal names used in the CE1921 ARMv4 central processing unit circuit. Color codes are used to help you see the instructions control signals moving through the pipelined and getting "hotter" as they get closer to completing the instruction. Arithmetic flow feedback paths are drawn downward while control flow feedback paths are drawn upward. Signal busses are shown as larger wires than single signals. Clock and reset signals are implied by the registers.

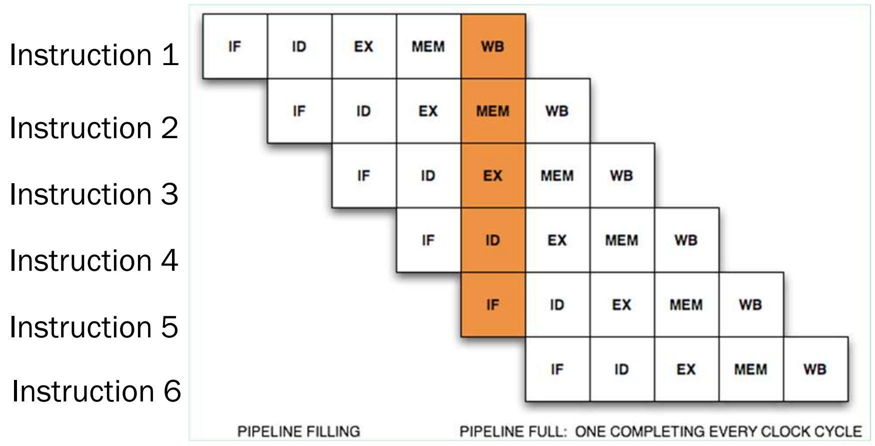
The full pipelined processor microarchitecture was drawn on the whiteboard in lecture. This diagram collapses the components of each stage into a black box labeled with the stage name. For example, the instruction ROM, PC+4 adder, and PC+8 adder have been collapsed into the FETCH black box symbol. Similarly, the source multiplexer the ALU, and the CPSR have been bundled into the execute stage.

Here are some key points about the pipelined signals:

- Signals that do not leave a stage do not pipeline. For example, the controller creates REGDST to choose the A2 address in decode. This signal does not pipeline. Similarly, the EXTS signal is used in decode and does not pipeline. On the other hand, REGWR is needed by the writeback stage and thus IDEX, EXMEM, and MEMWB must all sample this signal. As a signal pipelines, its name changes to reflect the stage the instruction is in. In lecture, REGWR represented the controller command in decode, REGWRE represented the instruction in execute, REGWRM represented the instruction in memory, and REGWRW represented the instruction in writeback.

- Like REGWR, A3 must pipeline the entire length of the pipeline because the destination address is ***part of the instruction*** and the instruction doesn't finish until writeback.
- PC+8 pipelines through IFID in our classroom implementation because the branch address adder, the extender and the controller were placed in the decode circuit.
- RD2 is the memory data in STR instructions. Thus, it must pipelined through IDEX and EXMEM.

PIPELINED INSTRUCTIONS



This diagram illustrates instruction level parallelism. Time is the horizontal axis while instructions are plotted on the vertical axis. Each circuit stage block represents one clock period. The fetch circuit retrieves the next instruction (PC+4) on each clock period. After five instructions, the pipeline will be ***full*** and, as long as the pipeline stays full with valid instructions, an instruction completes on each clock period. Virtually, it appears that an instruction completes every clock cycle – even though that is not the real case. It takes five clock periods, but because multiple instructions are in flight, there is no longer wasted delay.

In the single cycle circuit, the clock period was set to the summation of all stage delays: $\tau_{SCP}=\Delta_F+\Delta_D+\Delta_E+\Delta_M+\Delta_{WB}$

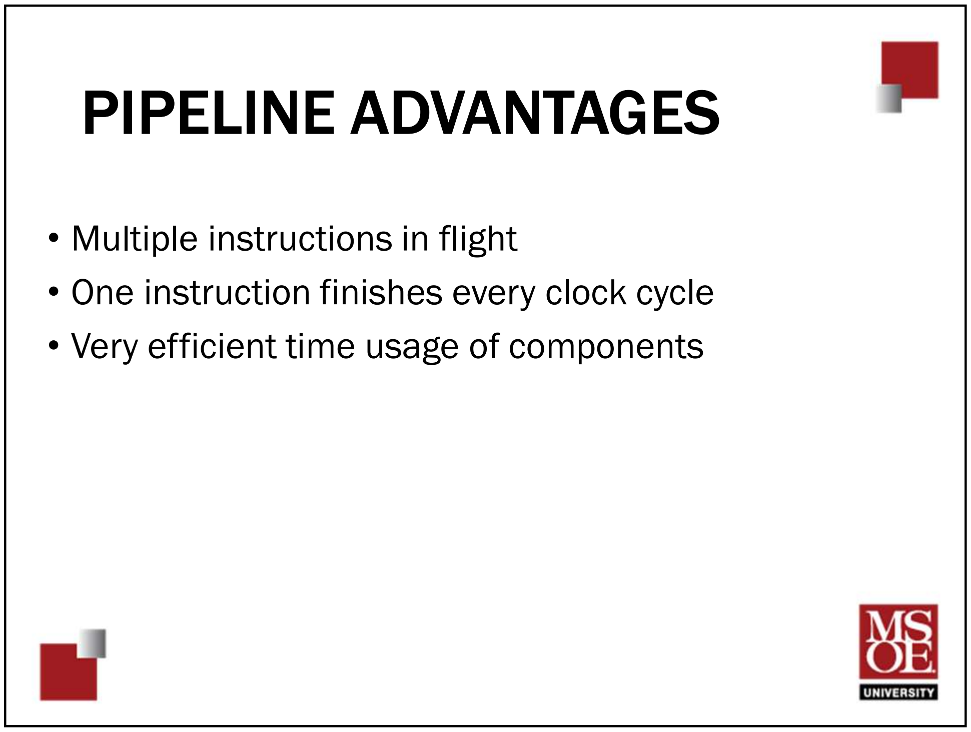
In the pipelined processor, the clock period is reduced to be the memory stage delay: $\tau_{PIPE}=\Delta_M$

Ideally, the new period is one-fifth of the original. In this ideal case, the full pipeline is completing an instruction 5 times faster than the single cycle processor.

Consider this example. Suppose a one million instruction program executes on a single-cycle processor with a clock period of 5μs. This program will require 1E6 instructions * 5E-6 s / instruction = 5s.

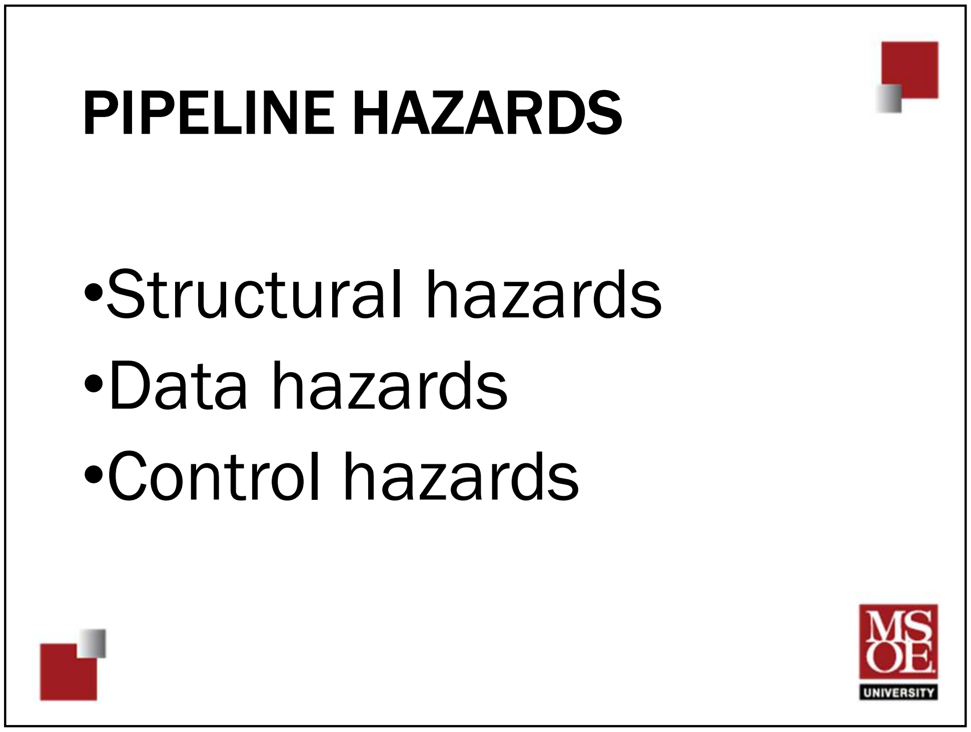
Now, if the pipelined version can achieve the ideal clock period of 5μs/5 stages = 1μs, then the new execution time assuming a full pipeline for all one million instruction is 1E6 * 1E-6 = 1s.

This shows that pipelining improves performance as long as the pipeline remains full.



PIPELINE ADVANTAGES

- Multiple instructions in flight
- One instruction finishes every clock cycle
- Very efficient time usage of components

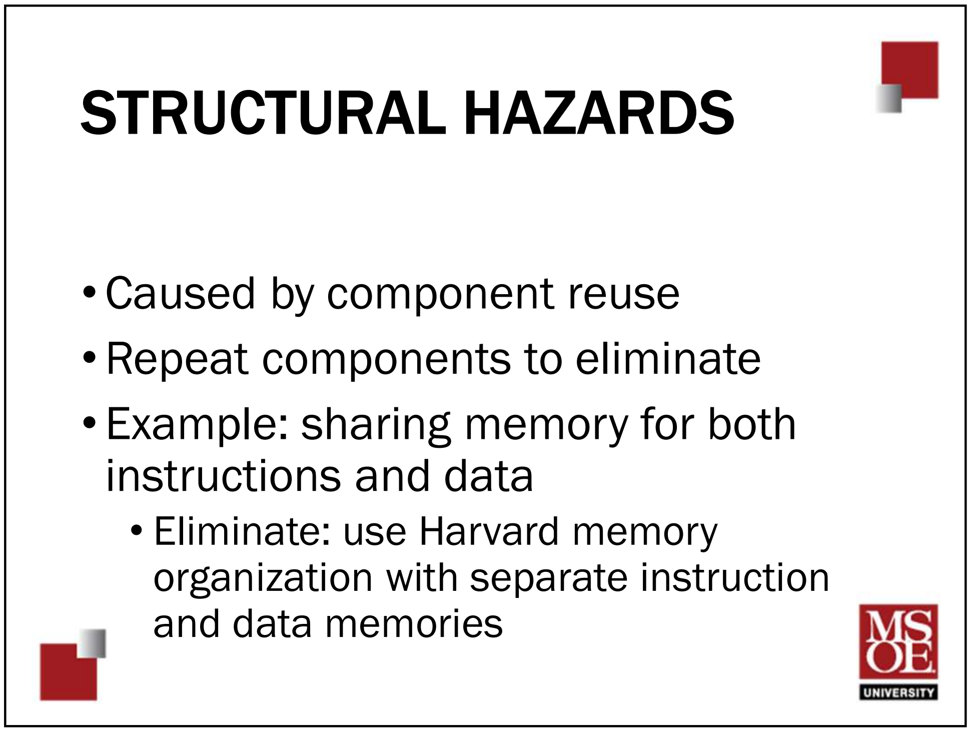


PIPELINE HAZARDS

•Structural hazards
•Data hazards
•Control hazards

Unfortunately, keeping the pipeline full is a hard task. We know that it is important to keep it full to improve performance. Things that prevent us from keeping the pipeline full are called **hazards**. Hazards fall into three categories:

- Structural exist when the circuit itself naturally limits performance.
- Data hazards exist when programmers regularly reuse destination registers in very nearby assembly language instructions as source data values.
- Control hazards exist because conditional branches are natural in algorithms.

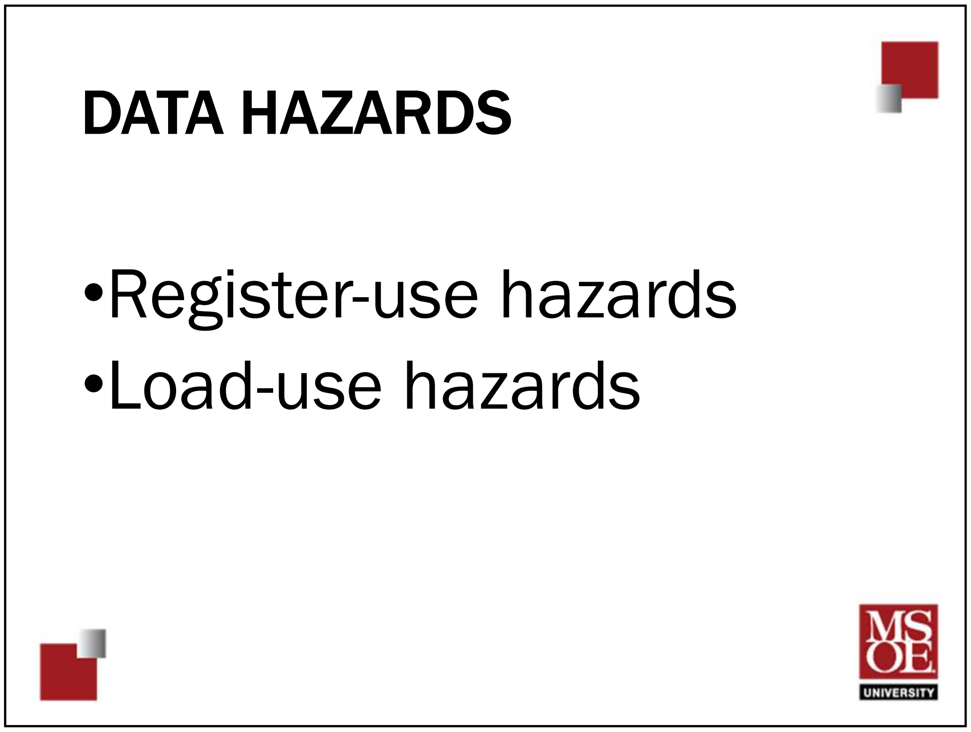


STRUCTURAL HAZARDS

- Caused by component reuse
- Repeat components to eliminate
- Example: sharing memory for both instructions and data
  - Eliminate: use Harvard memory organization with separate instruction and data memories

Structural hazards are most often caused because the circuit fails to replicate circuitry and instead relies on a common circuit component to do multiple tasks during computation.

- One example is the shared instruction and data memory. The address presented to the memory can only represent either the data address or the instruction address in every clock cycle, but not both. To eliminate the hazard, the Harvard organization is used rather than the Princeton.
- Another example is the arithmetic circuitry that calculates the branch address. If these adders are not included, and instead the calculations relied on the ALU, then they could not occur at the same time that the instructions arithmetic is occurring.



DATA HAZARDS

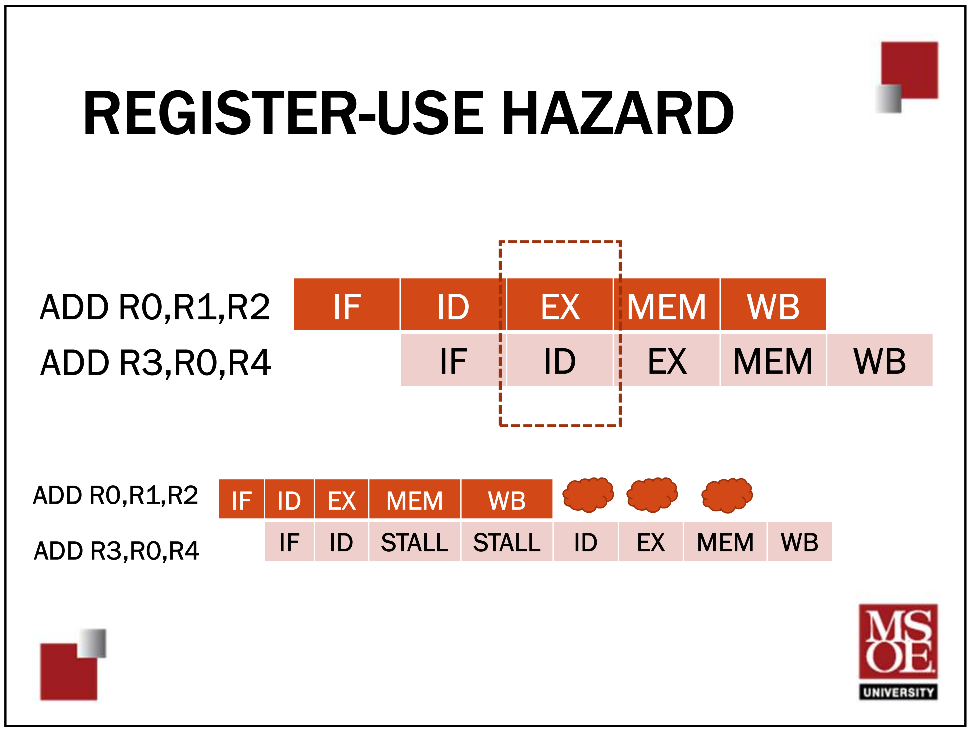
- Register-use hazards
- Load-use hazards

Data hazards exist when an assembly language statement **uses the destination register of an earlier instruction before that earlier instruction has finished executing through the pipeline.**

Two types of use can occur:

- The assembly language statement can use the destination register of an arithmetic instruction: ***register-use*** hazard.
- The assembly language statement can use the destination register of a memory load instruction: ***load-use*** hazard.

Both hazards exist are possible because the instruction in the decode stage needs the writeback data from **execute** or **memory**.



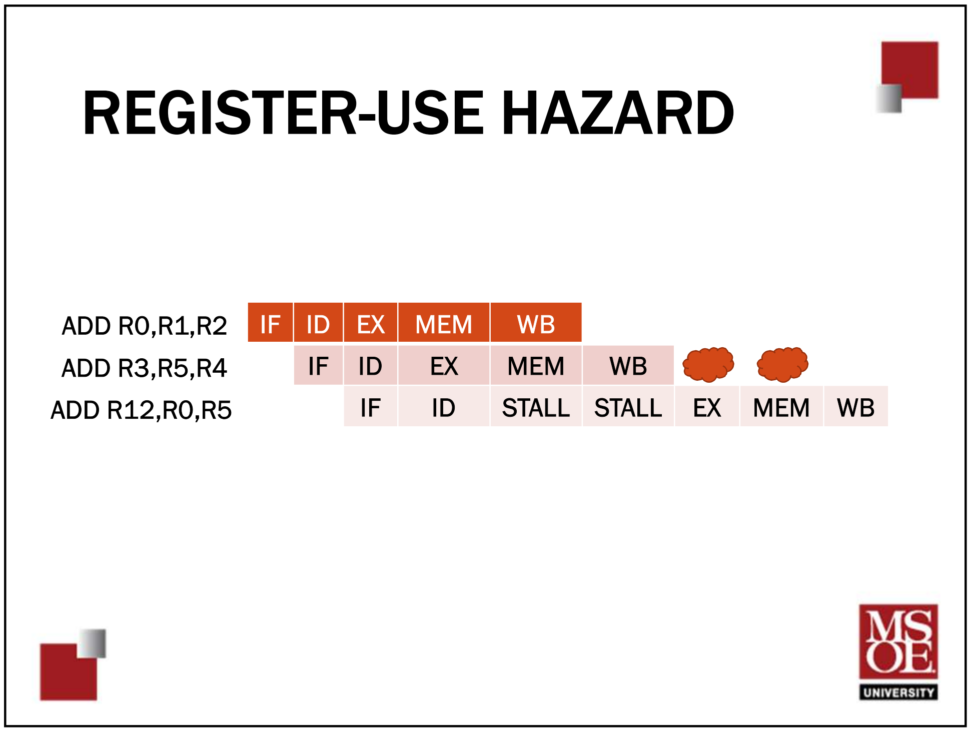
REGISTER-USE HAZARD

ADD R0,R1,R2	IF	ID	EX	MEM	WB				
ADD R3,R0,R4		IF	ID	EX	MEM	WB			

ADD R0,R1,R2	IF	ID	EX	MEM	WB				
ADD R3,R0,R4		IF	ID	STALL	STALL	ID	EX	MEM	WB

Consider the two instructions shown in the two diagram.

- The dashed rectangle illustrates the register-use hazard at Rn in the second instruction. In that instruction, **Rn is equal to the destination register Rd** of the instruction immediately in front of it.
- The pipeline must stall because the value of R0 is not stored I the register file when the second instruction needs it.
- Writeback periods that do not have valid write-backs are indicated in the second diagram with bubble symbols..
- In this example, the distance between the instruction with Rn = R0 and the instruction with Rd = 0 is one. **When the distance is one, three bubbles are inserted.**
- In our class, we have used a single clock edge (rising) to advance all registers in the processor. If we had chosen to sample the register file on the falling edge, then the decode circuit could complete decode in the second have of the writeback clock cycle. This would result in only two bubbles inserted rather than three.



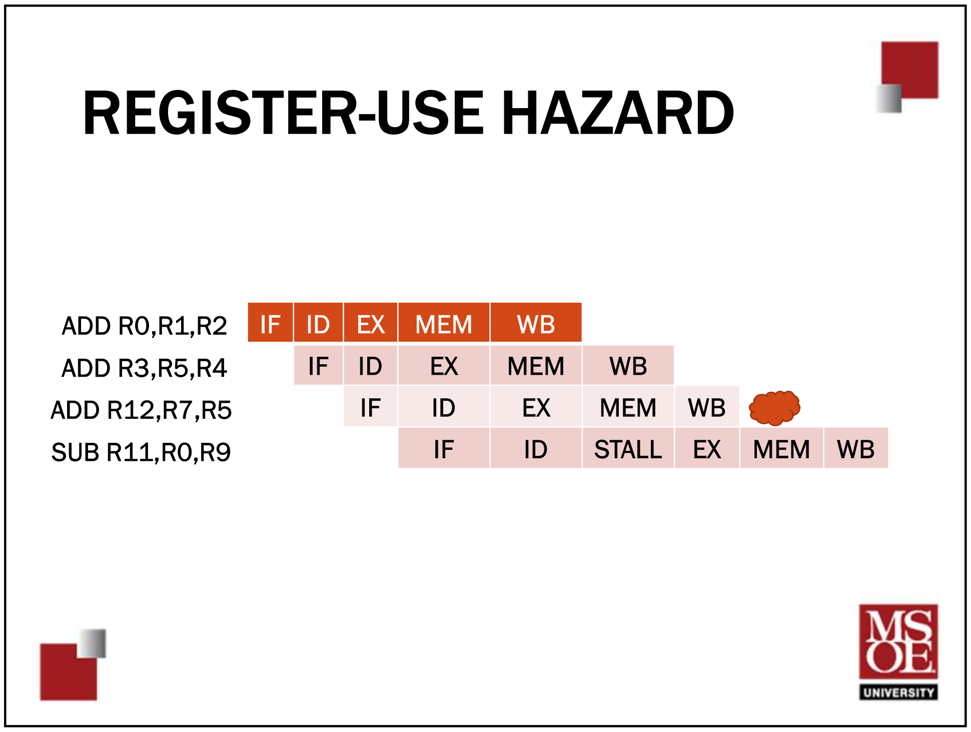
REGISTER-USE HAZARD

ADD R0,R1,R2	IF	ID	EX	MEM	WB				
ADD R3,R5,R4		IF	ID	EX	MEM	WB			
ADD R12,R0,R5			IF	ID	STALL	STALL	EX	MEM	WB

Consider the three instructions shown in the two diagram.

tw

- In this example, the distance between the instruction with Rn = R0 and the instruction with Rd = R0 is two. **When the distance is two, two bubbles are inserted.**

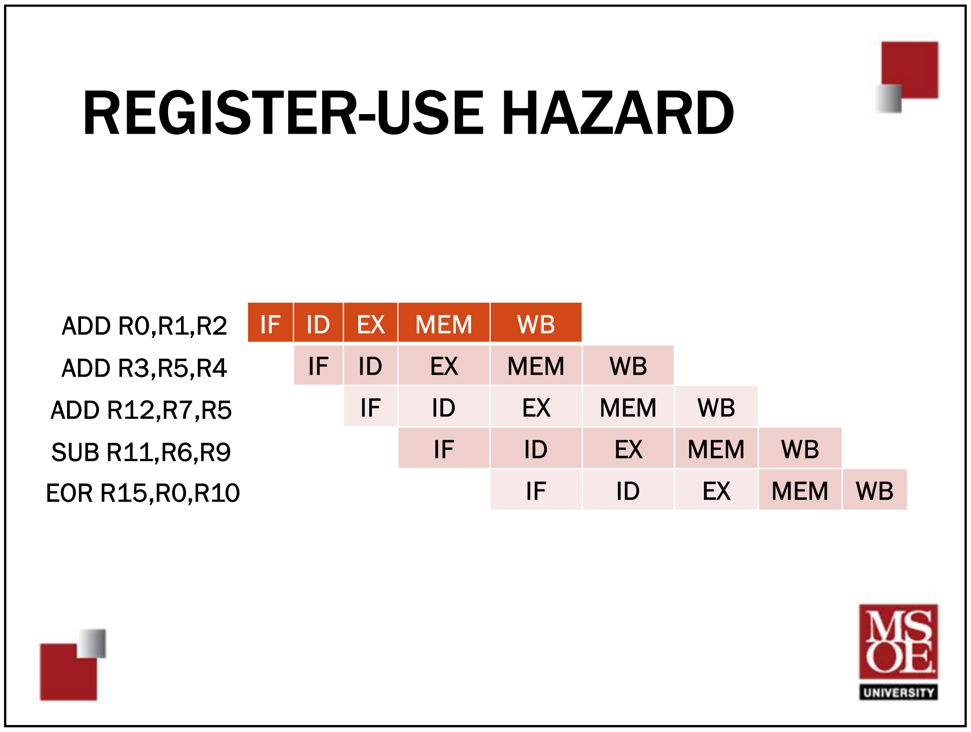


REGISTER-USE HAZARD

ADD R0,R1,R2	IF	ID	EX	MEM	WB				
ADD R3,R5,R4		IF	ID	EX	MEM	WB			
ADD R12,R7,R5			IF	ID	EX	MEM	WB		
SUB R11,R0,R9				IF	ID	STALL	EX	MEM	WB

Consider the three instructions shown in the two diagram.

- In this example, the distance between the instruction with Rn = R0 and the instruction with Rd = R0 is three. **When the distance is three, one bubble is inserted.**



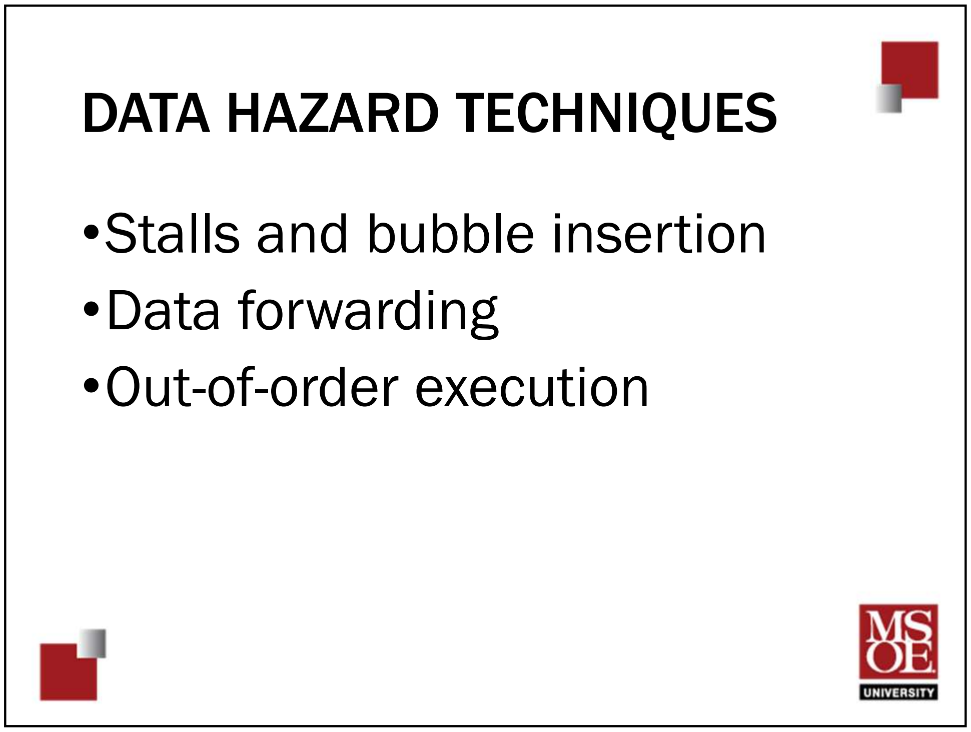
REGISTER-USE HAZARD

ADD R0,R1,R2	IF	ID	EX	MEM	WB				
ADD R3,R5,R4		IF	ID	EX	MEM	WB			
ADD R12,R7,R5			IF	ID	EX	MEM	WB		
SUB R11,R6,R9				IF	ID	EX	MEM	WB	
EOR R15,R0,R10					IF	ID	EX	MEM	WB

Consider the three instructions shown in the two diagram.

- In this example, the distance between the instruction with Rn = R0 and the instruction with Rd = R0 is four. **When the distance is four, no bubbles are inserted.**

The number of bubbles inserted is equal to four minus the distance. **Bubbles = 4 – hazard distance**, where the hazard distance is a positive integer in the set {1, 2, 3, 4}.

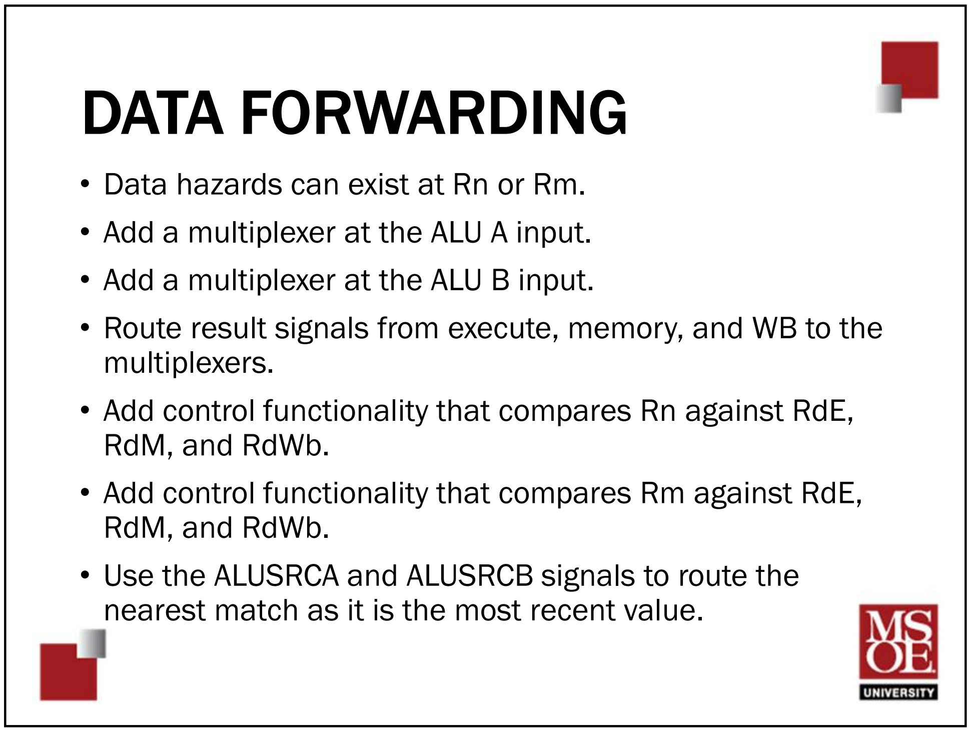


DATA HAZARD TECHNIQUES

- Stalls and bubble insertion
- Data forwarding
- Out-of-order execution

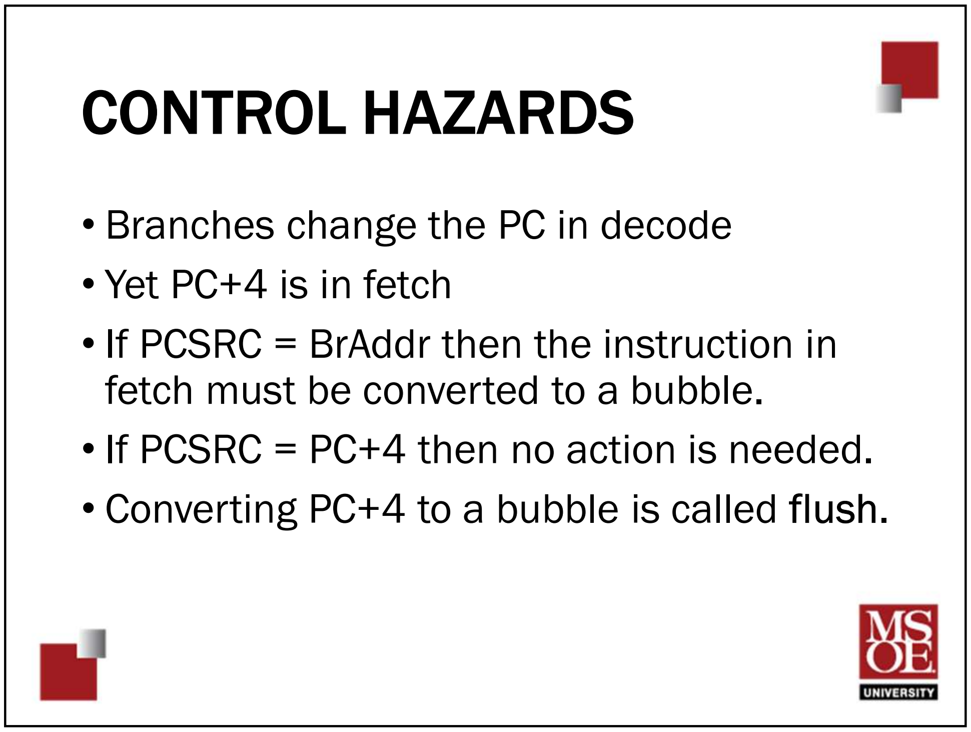
Computer architects worked for many decades to create different solutions for data hazards that could be evaluated for efficiency and overall performance.

- The simplest technique to handle the data hazard is to stall the pipeline. Stalls introduce bubbles, however, which reduce the overall throughput as instructions stop writing back for a few periods of time. **Stalls are undesirable**. In fact, a good rule of thumb for pipeline designers is ***every bubble counts***. This reminds designers to work to eliminate or minimize bubbles.
- Data forwarding provides the write-back data early using feedback lines from later stages to multiplexers in front of ALU A and ALU B.
- Out-of-order execution finds instructions that can be inserted into stall slots without changing the program result. Both compile-time techniques and run-time techniques exist. These are advanced algorithms discussed in the Computer Architecture 2 elective. Interested students can reference Tomasulo's algorithm and the scoreboard technique.



DATA FORWARDING

- Data hazards can exist at Rn or Rm.
- Add a multiplexer at the ALU A input.
- Add a multiplexer at the ALU B input.
- Route result signals from execute, memory, and WB to the multiplexers.
- Add control functionality that compares Rn against RdE, RdM, and RdWb.
- Add control functionality that compares Rm against RdE, RdM, and RdWb.
- Use the ALUSRCA and ALUSRCB signals to route the nearest match as it is the most recent value.

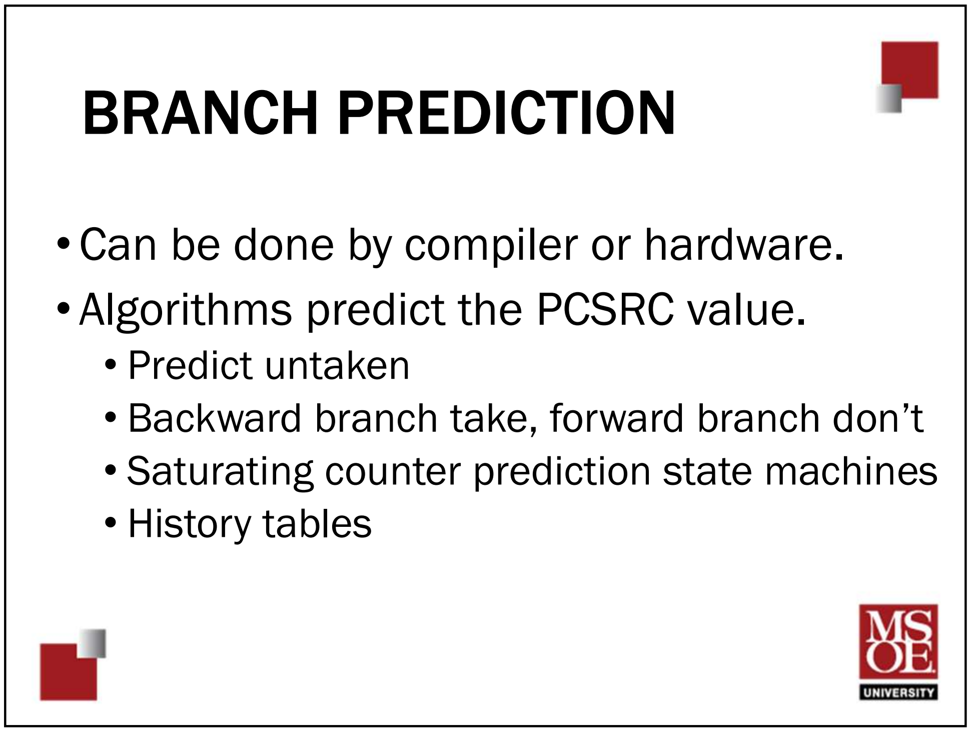


CONTROL HAZARDS

- Branches change the PC in decode
- Yet PC+4 is in fetch
- If PCSRC = BrAddr then the instruction in fetch must be converted to a bubble.
- If PCSRC = PC+4 then no action is needed.
- Converting PC+4 to a bubble is called flush.

Control hazards occur when branches adjust the program counter to something besides PC+4. Like data hazards, computer architects have spent decades working toward solutions.

- The simplest technique is to insert a bubble by converting the instruction in fetch to a **no-operation (NOP)**. This operation is called a **flush**. This introduces one bubble in the pipeline. A classic way architects plan for flush is by ensuring that machine code binary number of 0 results in no change to the stored state in the register file, memory, and status register. In ARM, an all 0 machine corresponds to the instruction ANDEQ R0,R0,R0. When any number is “anded” with itself, the same number results. Thus, there would be no change in the state of R0.

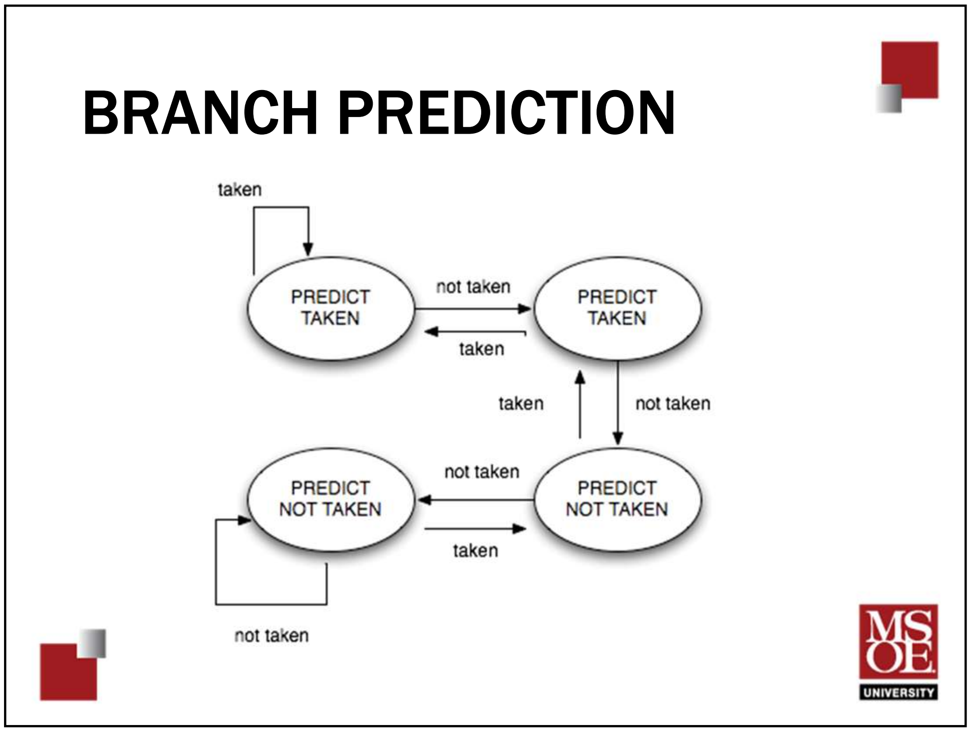


BRANCH PREDICTION

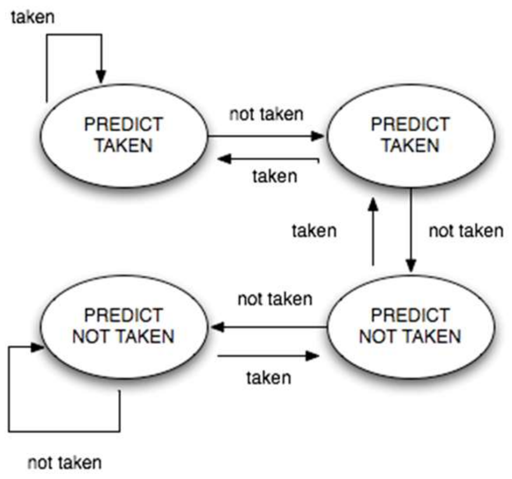
- Can be done by compiler or hardware.
- Algorithms predict the PCSRC value.
  - Predict untaken
  - Backward branch take, forward branch don't
  - Saturating counter prediction state machines
  - History tables

Branch prediction is a control hazard technique designed to eliminate the flush is branch prediction. Many different algorithms have been explored.

- Predict untaken is just defaulting to PC+4 and inserting stalls when needed. It is the least effective technique at avoiding stalls.
- Most loops branch backward. By comparing the sign of the branch constant (immediate) in fetch, a predictor can choose to set PCSRC = BrAddr if the constant is negative, suggesting backward and likely a loop that will execute many times.
- Many advanced techniques that use history tables of saturating counter prediction state machines – one per branch instruction – exist. These techniques are subjects in the Computer Architecture 2 elective course.
- After decades of research, modern advanced techniques achieve 97% prediction accuracy on benchmark programs.



BRANCH PREDICTION



This 2-bit branch predictor state machine shown in this picture has been shown to be rather effective. It does not require substantial silicon space. As a four state machine, it requires two state bit flip-flops as well as the logic to advance it between states based on the actual PCSRC signal created by the controller. It is a saturating counter state machine with strengthening prediction as the counter nears each end point. In other words, at each endpoint, there is a ***strong prediction*** while the similar state in the middle of the machine is a **weaker prediction**.