EECS151/251A Discussion 9

Christopher Yarp

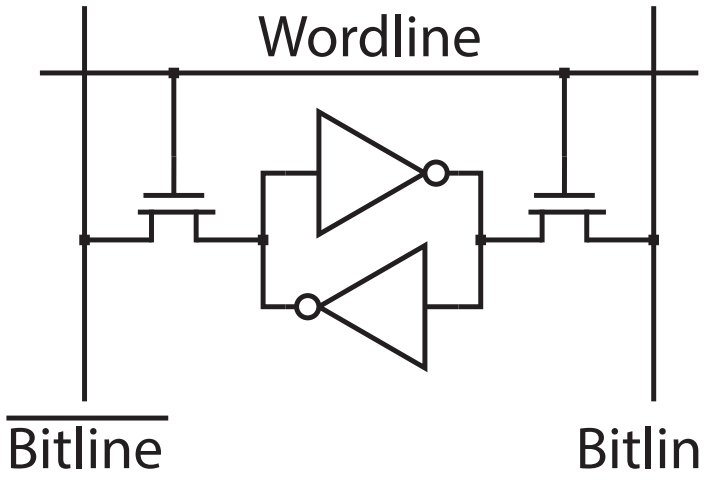
Apr. 5, 2019

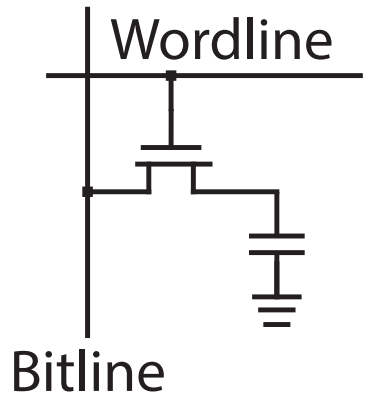
Plan for Today

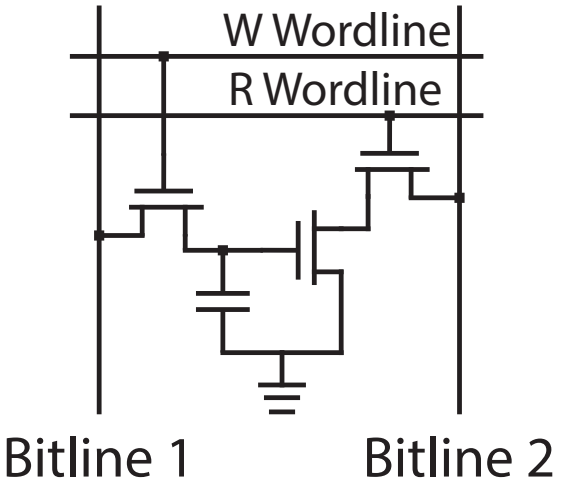
- Practice Problem
- Processor Review
- Your Questions

Recall Different Memory Memory Technologies

- 6T SRAM: Crossed coupled inverters (4 transistors) + 2 access transistors
- 1T DRAM: Data stored in capacitor. 1 transistor to write and read value. Read is a destructive operation
- 3T DRAM: Data is stored in transistor capacitance. 2 transistors to read, 1 to write. Read is non-destructive.







Memory Technology Practice Problem

- A device requires a 2 MiB (1MiB = 1024*1024 bytes) cache
- The cache line is 32 32-bit words
- The silicon foundry you are using includes 6T SRAM cells and 3T DRAM cells in the design kit for the CMOS logic process you are planning to use to implement your processor.

1. You want to have an equal number of rows and columns in the memory. How many rows and columns do you need?
2. What is the approximate size difference between an SRAM version of the memory and the DRAM version of the memory (assume all transistors are the same size and ignore the periphery logic).
3. What are some of the principle disadvantages of the DRAM cell compared to the SRAM cell?
4. Why do you suspect the silicon foundry did not provide a 1T DRAM cell for this CMOS logic process?

Number of Rows & Columns

- We want an equal number of rows and columns if possible.
- The number of bits in the memory is 2 MiB = 2*1024*1024*8 = 1024^2 * 4^2.
- The number of columns & rows is ideally sqrt(1024^2 * 4^2) = 4096
- The cache lines are 32*32 = 1024 bits wide
- 1024 fits evenly into 4096: 4096 is an acceptable number of columns

- Rows: 4096
- Columns: 4096

Area Comparison

- The 3T DRAM requires 3 transistors
- The 6T SRAM requires 6 transistors
- If we neglect the size of the transistors and the difference in wiring, the 3T DRAM cell is approximately ½ the size of the 6T SRAM cell
- This would lead to an array that is approximately ½ the size

- Note: this is a back of the envelope approximation. SRAM and DRAM are wired differently which will have impacts on the size of the cells and the periphery logic.
- However, DRAM can generally be more dense than SRAM

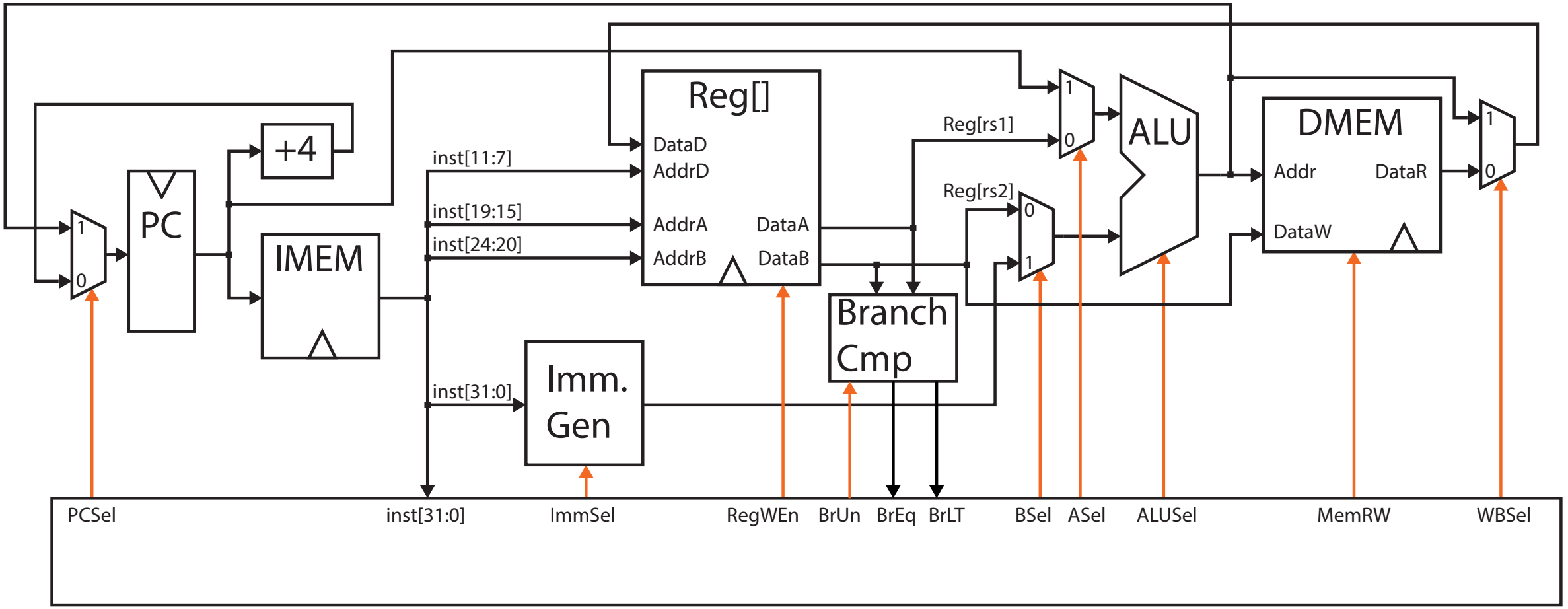
Disadvantages of 3T DRAM

- Even through the 3T DRAM does not have destructive read operations (like 1T DRAM), leakage still exists.
- Charge will slowly leak off the capacitor, requiring values to be periodically refreshed.
- DRAM is also non-restoring while SRAM is restoring
  - Makes DRAM more sensitive to noise and other non-idealities

Why is the 1T DRAM Unavailable?

- 1T DRAM requires larger capacitance in the cell than what can be provided in a digital logic focused process.
- Special DRAM processes exist which allow large capacitors to be formed.
- These processes are typically not as well optimized for digital logic. This is why we don't typically use DRAM processes when implementing logic heavy ASICs.
- The integration of DRAM with logic is an ongoing research topic with 3D stacking of DRAM dies on top of CMOS dies being one option
  - Lookup HBM (High Bandwidth Memory) if you are interested

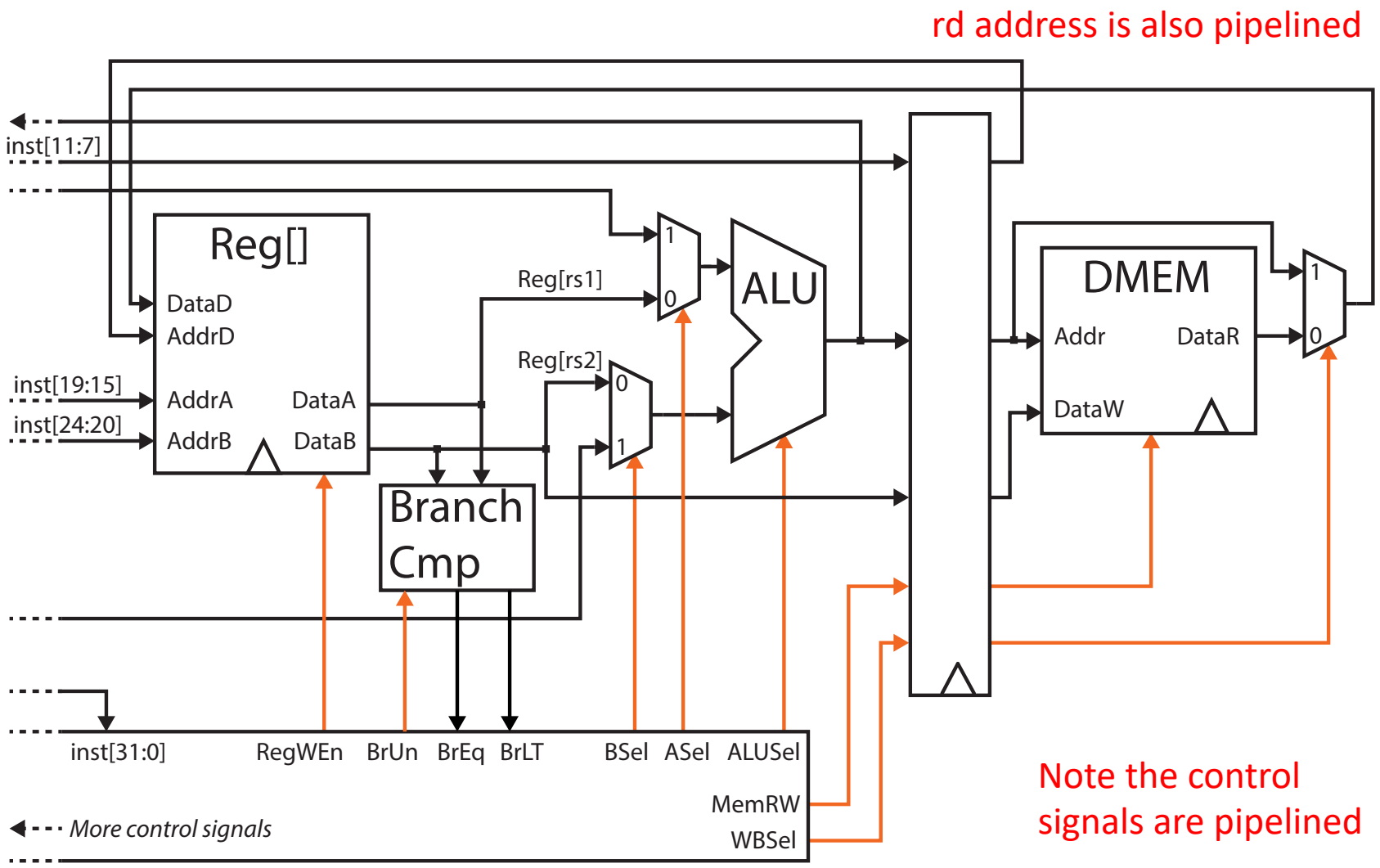
Quick Review – Single Cycle RISCV



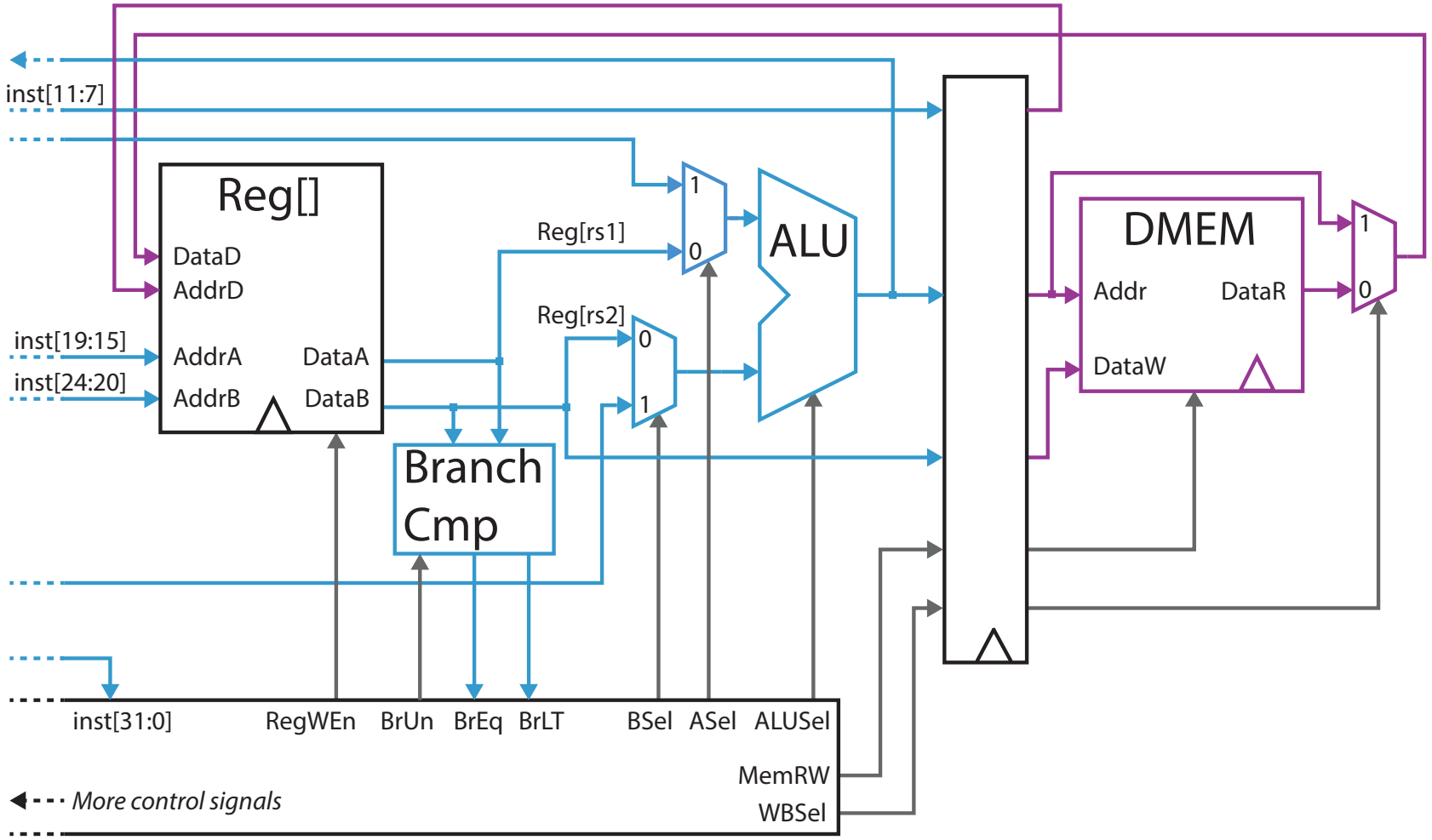
The 5 Stage Pipeline

- In general, most RISC Load/Store style processors need to accomplish 5 tasks when executing an instruction. Each task can be made into a separate pipeline stage (the 5 stage pipeline).
  - IF (Instruction Fetch): Fetch the Instruction from Memory
  - ID (Instruction Decode): Decode what the instruction does, set control signals for the instruction, fetch relevant values from the register file
  - EX or X (Execute): Execute the instruction if it is an arithmetic instruction (using the ALU). Compute addresses for memory load/store commands. Determine if a branch is taken or not.
  - MEM or M: Read from memory if the instruction is a load, write to memory if the instruction is a store
  - WB (Write Back, or Register Write): Write the value computed by the ALU (for arithmetic instructions) or a value read from memory (load instructions) into the register file.
- In the labs and in the HW, we ask for a 3 stage pipeline. Some of these functions need to me merged together

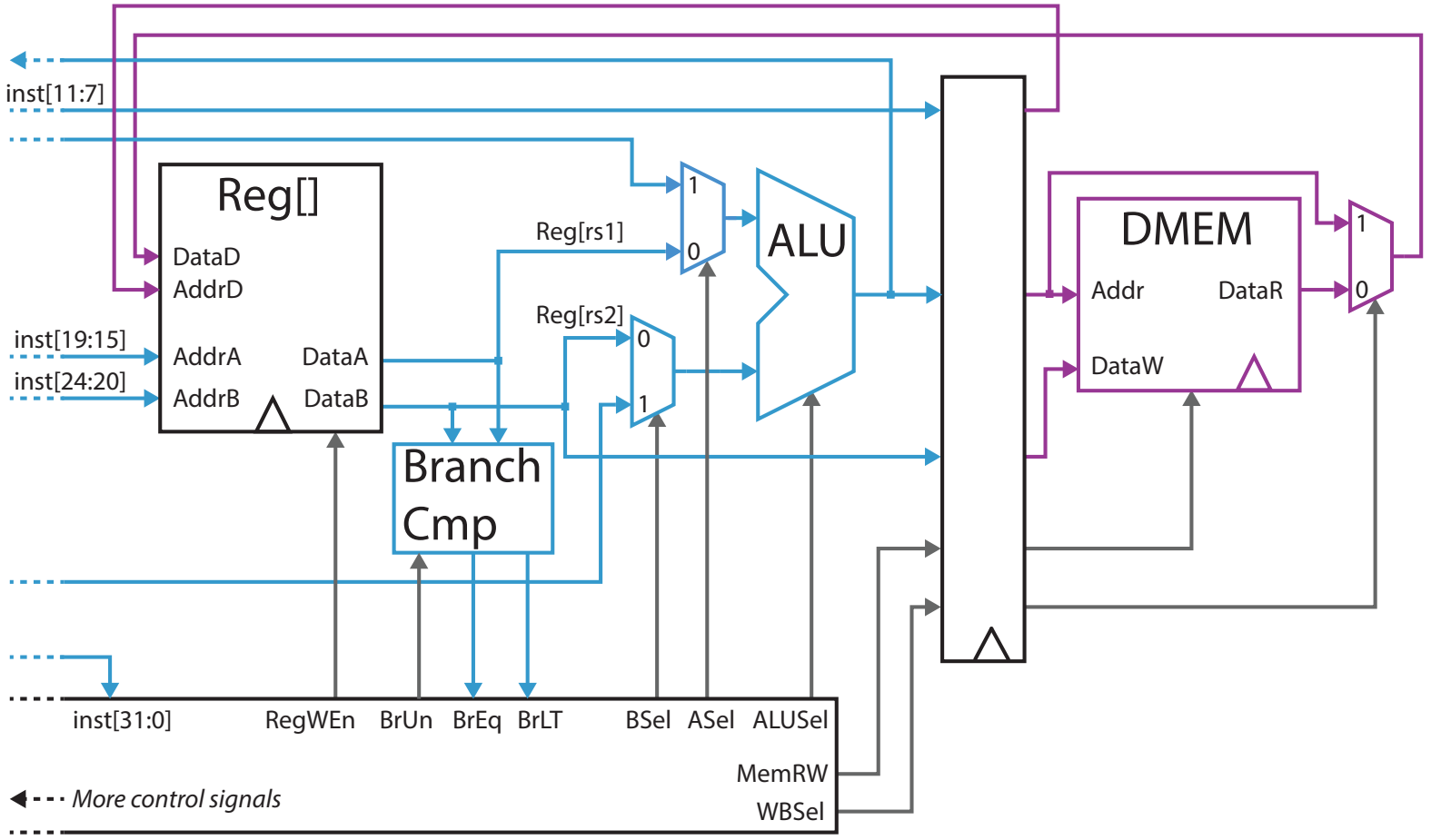
Pipelining for Speed



Pipelining for Speed



Data Hazard



sub r5, r3, r4

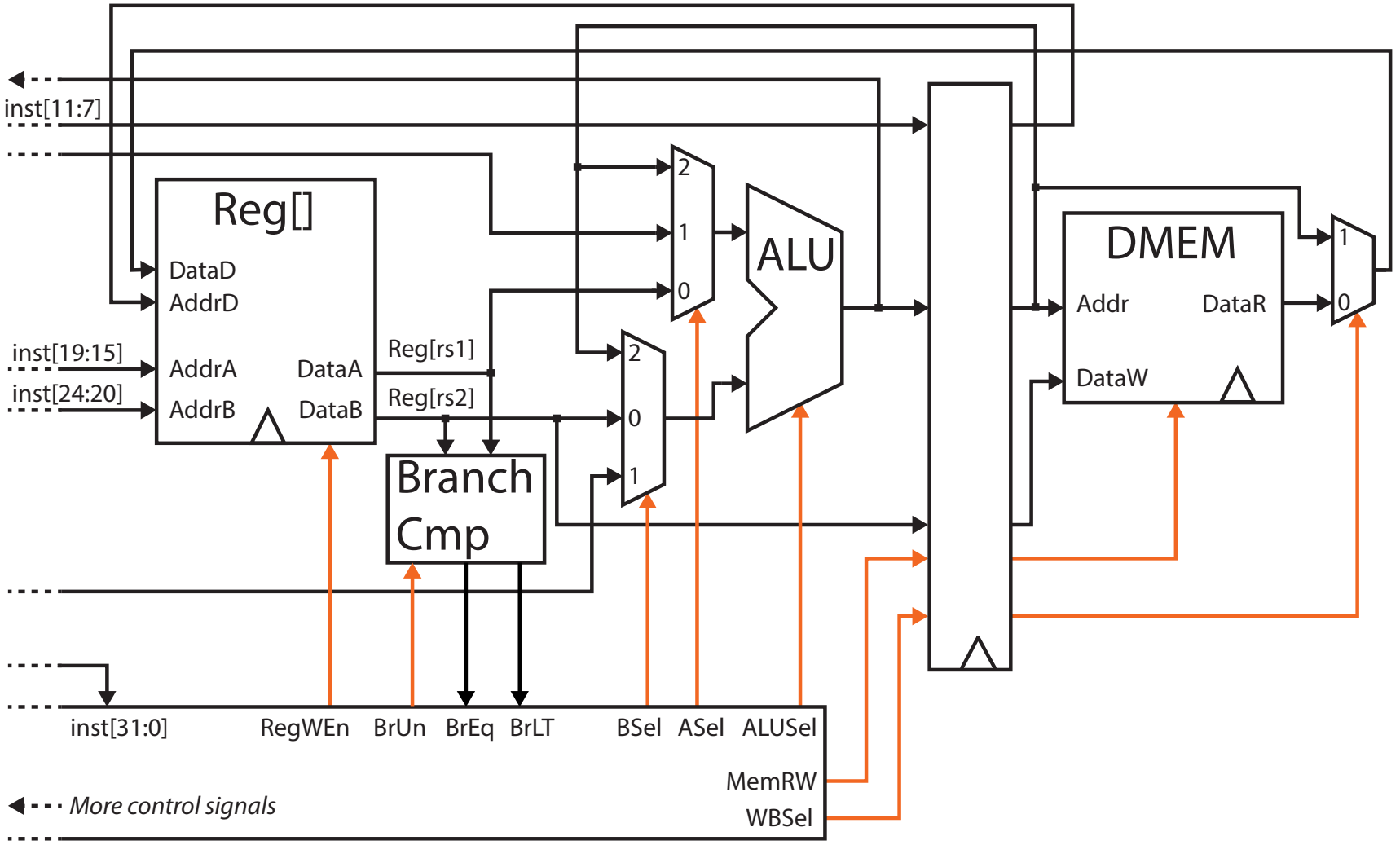
add r3, r1, r2

Data Hazard
The result of the add instruction will not be in the reg file when the sub instruction tries to read it

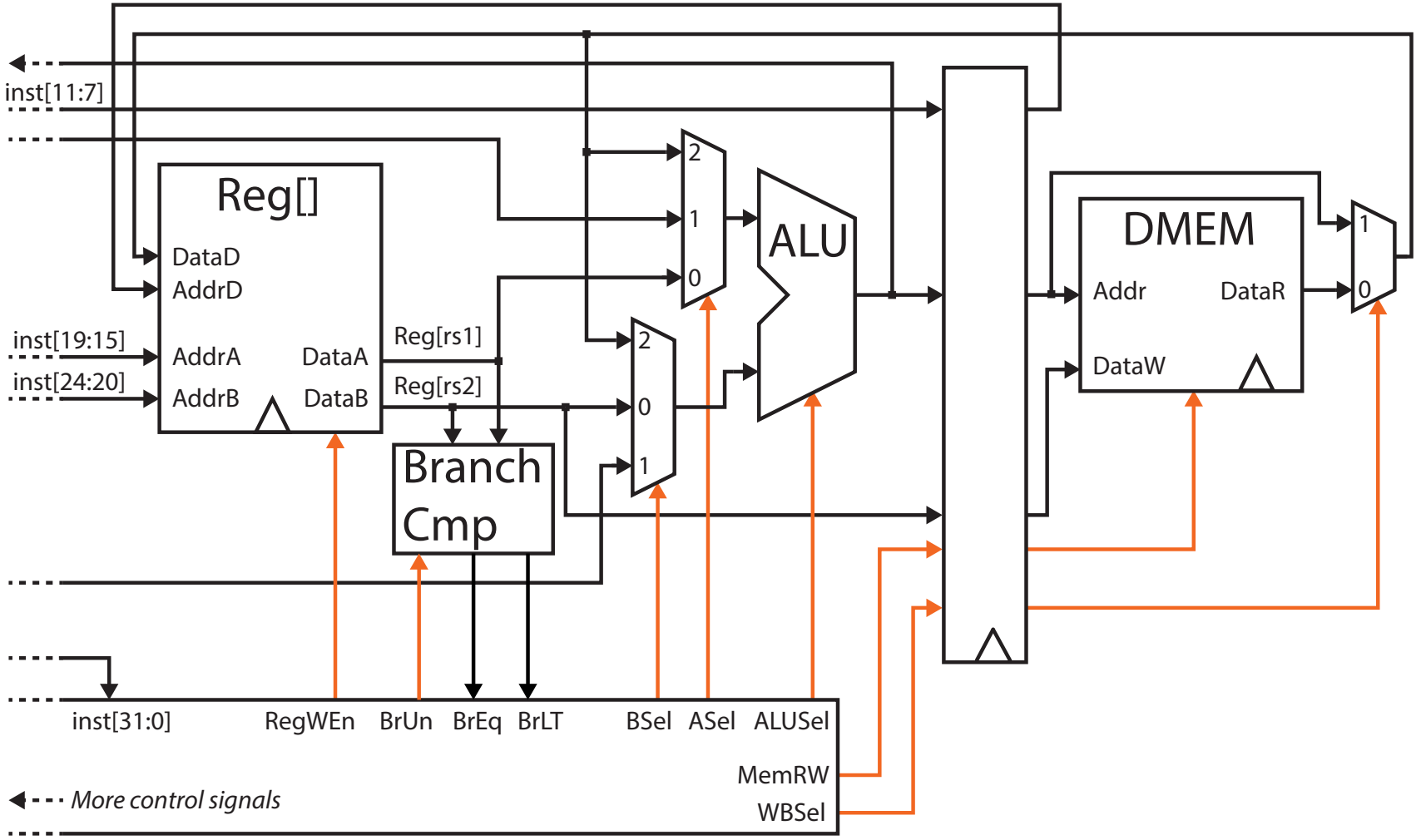
Data Hazards & Load Hazards

- 2 Main Options for Dealing with Data Hazards
  - Stalling: Stall the other pipeline stages of the processor while you wait for an another instruction to finish
    - Involves freezing the program counter and preventing new instructions from being written into pipeline registers
  - Forwarding: Send the required information back to an earlier pipeline stage (in parallel with it being written into the register file)
- Load hazards are similar to Data Hazards: the value read from memory will not be ready for the

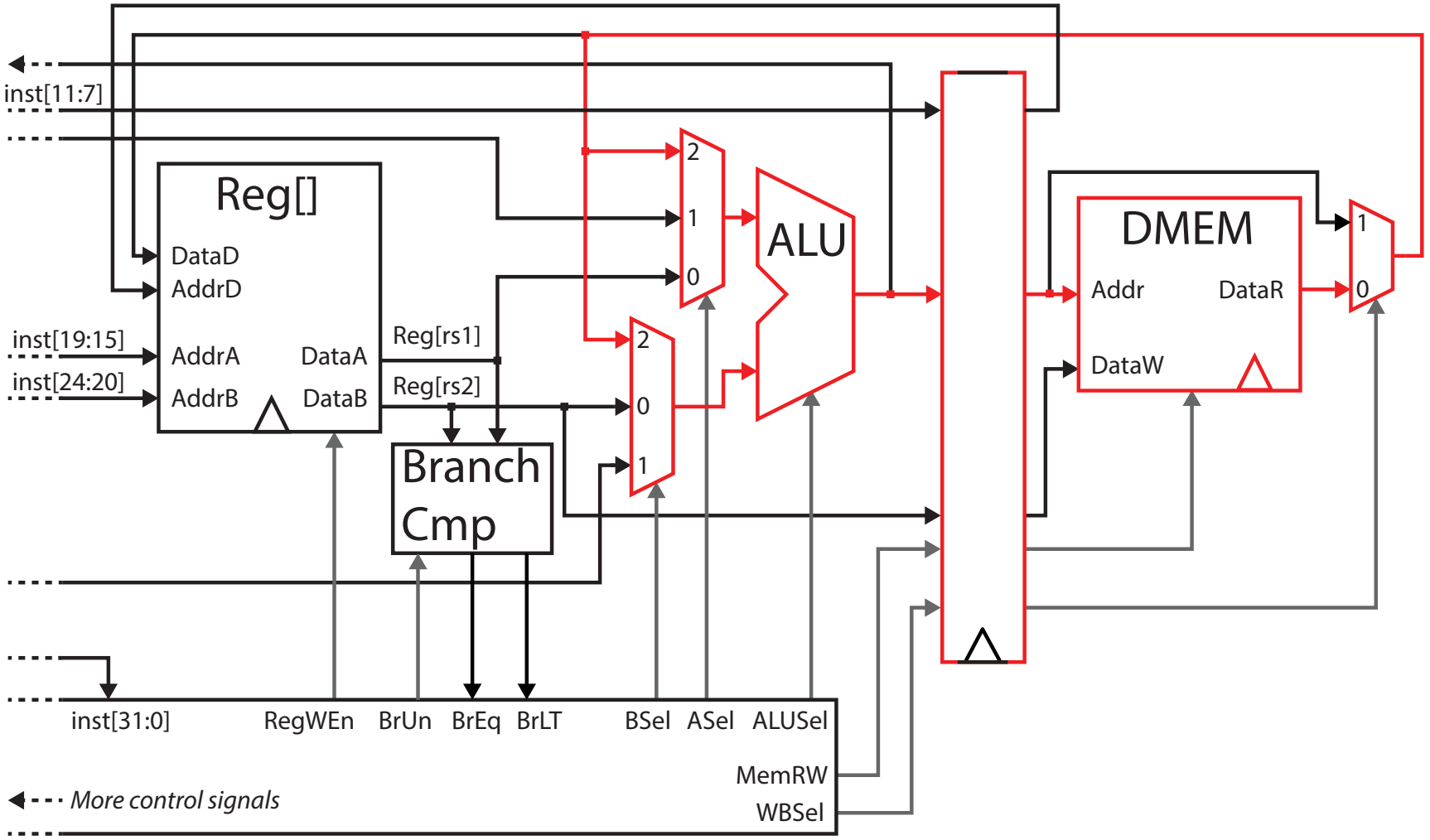
ALU Forwarding



Why don’t we forward memory loads too?



Forwarding memory results in a long critical path



Control Hazards

- We usually expect that the next instruction to be executed will be the next instruction in the program (PC+4).
- There are 2 main cases when this is not true:
  - Jump instructions: execution jumps to a specified instruction unconditionally
  - Branch instruction: execution jumps to a specified instruction if a condition is satisfied
- Where does the hazard come from?
  - In a pipelined processor, you may not know that an instruction is a jump or branch until another instruction has already entered the pipeline
    - You you may need to wait additional cycles for it to be determined if a branch is taken or to compute a target address
- How do we resolve the hazard?
  - We turn the instruction we fetched in error into a NOP (no operation) instruction
    - The most important thing is that this instruction should not modify any state (including the PC)
  - The NOP runs through the pipeline like any other instruction.

Tips for Implementing Processors

- Diagramming is a very important step in the design process
  - It helps you think through the major design decisions before getting lost in Verilog
- While it is easy to focus on the datapath of the processor, it is important to not neglect the control signals
- A common source of bugs in processor implementations in the lab come from the control logic
- Keep careful track of:
  - What control signals are needed?
  - When are they needed?
  - What information is required to correctly make control decisions?
  - Where/when does this information come from (the same or different pipeline stage)?
  - Is all the information available to make a decision?
    - Is a stall required?
    - Was a prediction made which needs to be validated or invalidated?