UF Herbert Wertheim College of Engineering
Department of Electrical & Computer Engineering
UNIVERSITY of FLORIDA

Dr. Christophe Bobda
Newman Waters
EEL3701C - Fall 2025

HW7: Advanced CPU Design

Consider the following processor, designed to be an extremely simplified version of a RISC (reduced instruction set computer) processor:

The instructions are now a bit larger – 16 bits, mainly to accommodate more registers in the processor, as well as the addition of a large memory component (RAM) where we can store data from calculations we perform.

The processor's 8 general-purpose registers are stored in a structure called a register file. A **register file** is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer.

Essentially, send in the 'index' of the two registers that you want to read to the component, and it will send out the data contained within those two. To write to a register, send in the 'index' of the register that you want to write to, send it data via the data line, and it will overwrite the specified register with the new data.

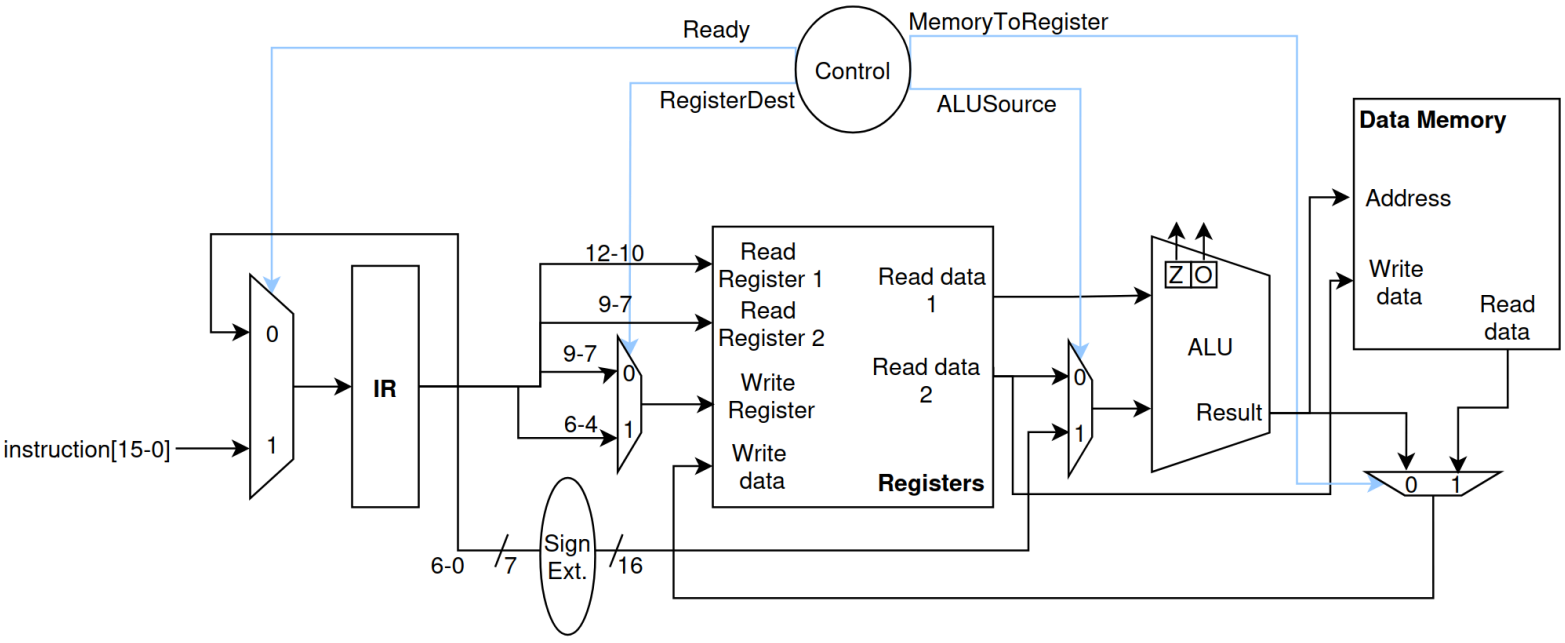


Figure 1: Basic 16-bit RISC processor with **8** 16-bit general purpose registers

UF Herbert Wertheim College of Engineering
Department of Electrical & Computer Engineering
UNIVERSITY of FLORIDA

Dr. Christophe Bobda
Newman Waters
EEL3701C - Fall 2025

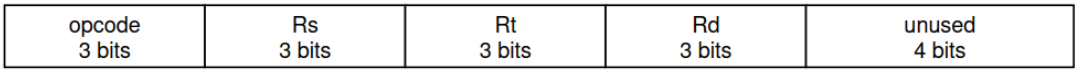
This datapath was created to implement the following instructions:

Operation Code	Mnemonic	Style	Description
000	Add Immediate, *ADDI*	Immediate	R[t] = R[s] + SignExtImm
001	Add, *ADD*	Register	R[d] = R[s] + R[t]
010	Load Immediate, *LDI*	Immediate	R[t] = SignExtImm
011	Load, *LD*	Register	R[t] = Mem[addr]
100	Store, ST	Register	Mem[addr] = R[t]
111	Stop, HALT	—	Stop processor

Table 1: Simplified Instruction Set

Note: Memory is word-addressed (each address contains one 16-bit value). Assume the add operations can handle signed numbers.

Register-Style



Immediate-Style

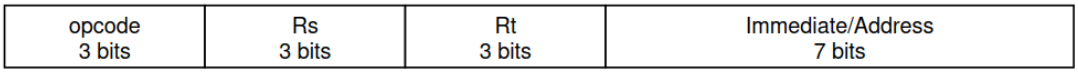


Figure 2: Basic Instruction Formats

Part 1:

a. Given a 16-bit instruction 0011110101010000, identify the instruction that is being performed. Explain what the instruction is doing (what registers are being accessed/modified/stored to, what style of instruction is this, etc.).

b. Why is the sign extension unit in *Figure 1* necessary for immediate instructions? Why would there be an issue if we didn't sign-extend the immediate/address?

c. For a *Load* instruction, what does the CPU need to do, step-by-step, to fetch a value from memory into a register based on *Figure 1*? What about the *Store* instruction?

Part 2:

Now, recall how tough it was to manually load in each instruction in Lab 6. This clearly is not the best method to program, since programs nowadays probably need hundreds, if not thousands of instructions to work correctly.

To automate this process, in place of an instruction register like in *Figure 1*, you will replace it with RAM. This will allow you to fill the RAM with instructions upon programming the processor, which will automatically run on each clock cycle.

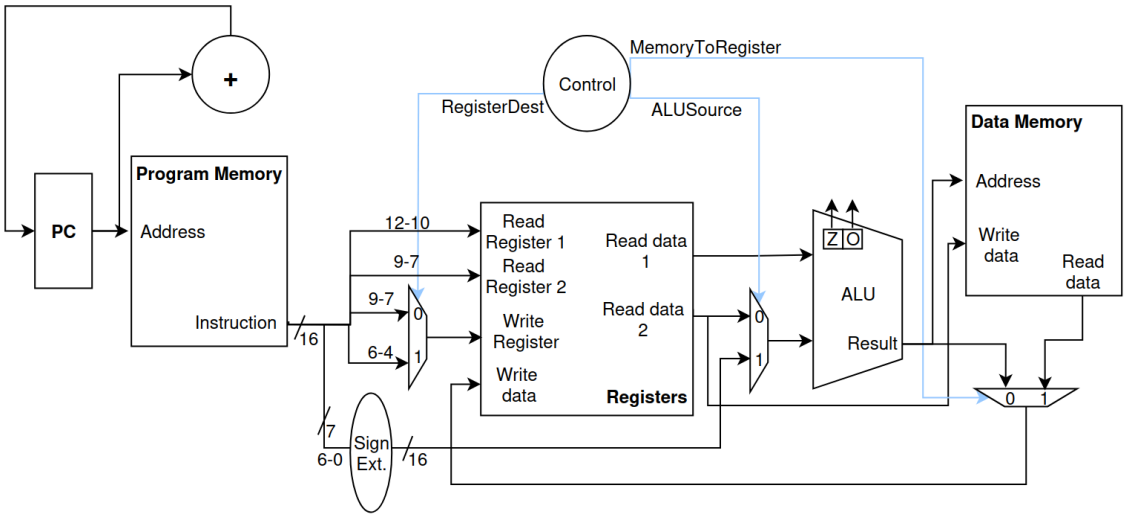


Figure 3: Instruction/Program Memory added to datapath

a. Based on the way the program memory will be laid out (1 16-bit instruction at each address), how much do you need to increment the program counter each clock cycle?

b. Thus, how many clock cycles does it take to execute a single instruction with this methodology? Why is this method not necessarily efficient, and how could we potentially improve execution time?

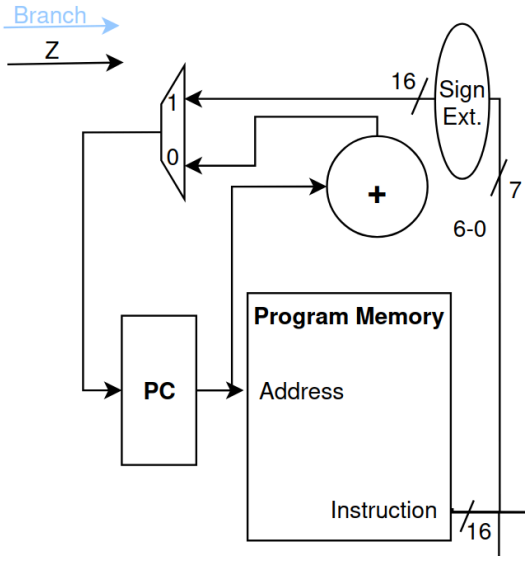
Part 3:
Now that we have our instruction memory integrated with our processor, we want to add some additional capabilities to allow us to jump to different addresses and perform conditional operations (jump somewhere in memory *if* some condition occurs).

Operation Code	Mnemonic	Style	Description
101	Branch if equal, *BEQ*	Immediate	if flag(Z) = 0, PC = addr
110	Jump, *JMP*	Immediate	PC = addr

Table 2: Added Branch and Jump Instructions

Note the new instructions above. The Z flag is outputted by the ALU when the previous instruction that executes equals 0. Example: ADDI R0, -5 – if R0=5, the operation adds 5 + (-5) = 0, which triggers the Z flag to be set (Z=1).

The controller sends out a control signal called *Branch* when the current instruction is a branch. You do not need to worry about how these are generated.



a. Inspect the following diagram, and based on the description above, combine the ***Branch*** control signal and the **Z** flag with some logic to choose between the processor going to the next instruction (Mux select 0), or the address to branch to (Mux select 1) (Draw out signals and logic gates/components).

b. The instruction set does not specify a comparison instruction. With our *very* limited instruction set, how would you perform a comparison operation to prepare for a branch instruction (conditional code)?

c. Based on the design of the instruction, how many addresses could we possibly jump to (i.e. how big should our Program/Instruction memory be at maximum)?

d. Explain the difference between conditional and unconditional jumps.

e. Can you think of any potential dangers of using the jump function (think of processor security)? What do you think would happen if a processor were to try to jump to a location outside of the normal address space on a real computer?

This processor was completely made up, based on the MIPS Single-Cycle implementation. If you want to understand how to do this more, try to design your own basic ISA and processor and create the hardware design in VHDL or Verilog. This will inherently be the best way to learn and see how the components work together!

Given that this is the last assignment, other than the final exam, we wanted to thank you for your hard work and dedication to learning both digital logic and computer organization with us. We hope that you enjoyed the course thoroughly and will see you in more computing classes in the future!!