RISC-V Datapath

Computer Architecture

Dr. Mahmoudi

May 6, 2024

Outline

-

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

Assembler

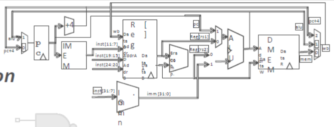
Machine Language Program (RISC-V)

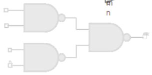
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

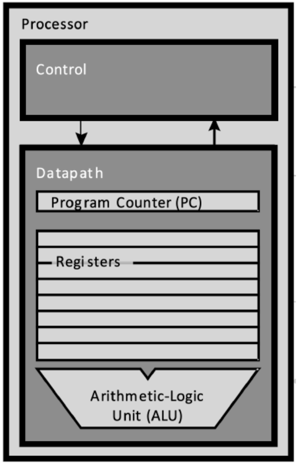
Logic Circuit Description (Circuit Schematic Diagrams)





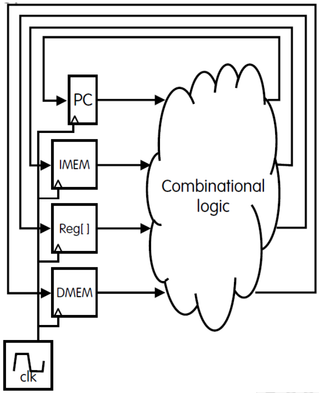
How do we build a Single-Core Processor?

- **Datapath ("the brawn"):** portion of the processor that contains hardware necessary to perform operations required by the processor.
- **Control ("the brain"):** portion of the processor that tells the datapath what needs to be done.



One-Instruction-Per-Cycle RISC V Machine

- The CPU is composed of two types of subcircuits: **combinational logic blocks** and **state elements**.
- On every tick of the clock, the computer executes one instruction:
  - Current outputs of the state elements drive the inputs to combinational logic.
  - whose outputs settle at the inputs to the state elements before the next rising clock edge.
- At the rising clock edge:
  - All the state elements are updated with the combinational logic outputs.
  - and execution moves to the next clock cycle.



Datapath elements

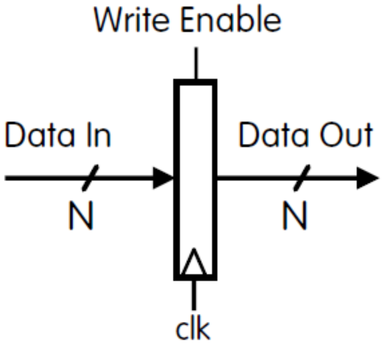
Datapath elements

- **Datapath element:** A unit used to operate on or hold data within a processor.
- In the RISC-V implementation, the datapath elements include:
  1. Instruction memory.
  2. Data memory.
  3. Register file.
  4. ALU.
  5. Adders.
  6. PC.

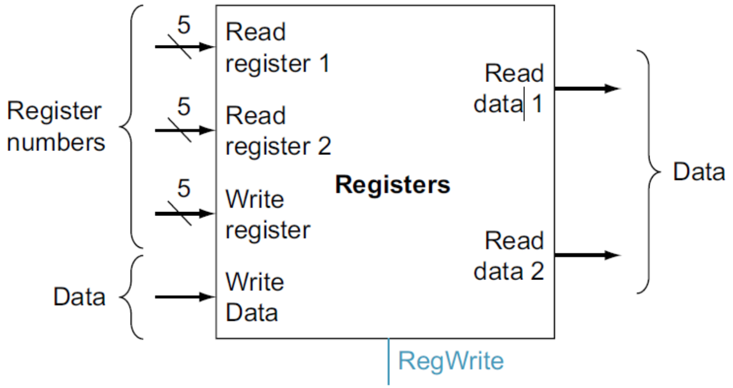
Building a Datapath: State Elements

program counter (PC), 32-bit register that holds the address of the current instruction.

- **Input:**
  - — N-bit data input bus,
  - — Write Enable “Control” bit (1: asserted/high, 0: deasserted/0)
- **Output:**
  - — N-bit data output bus.
- **Behavior:**
  - — If Write Enable is 1 on the rising clock edge, set Data Out=Data In.
  - — At all other times, Data Out will not change; it will output its current value.

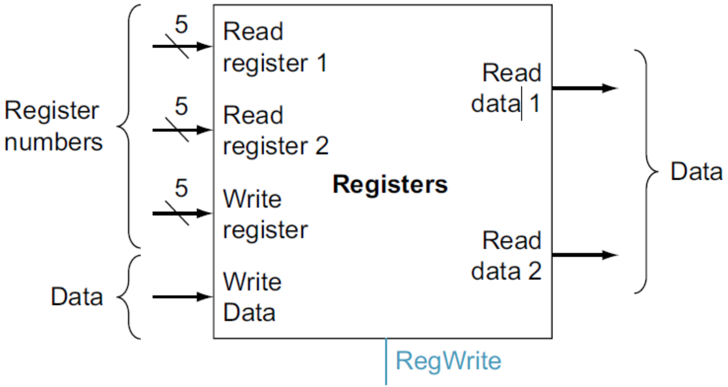


Building a Datapath: State Elements



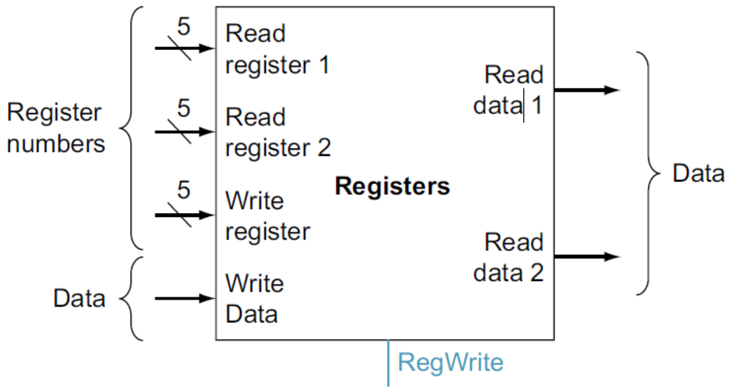
- The processor's 32 general-purpose registers are stored in a structure called a **register file**.
- A **register file** is a collection of registers in which any register can be read or written by specifying the number of the register in the file.

Building a Datapath: State Elements



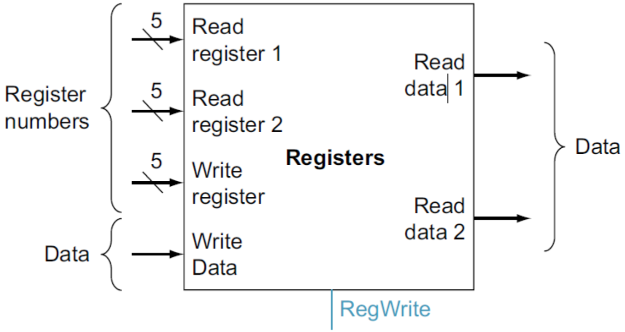
- For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers.

Building a Datapath: State Elements



- To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the data to be written into the register.

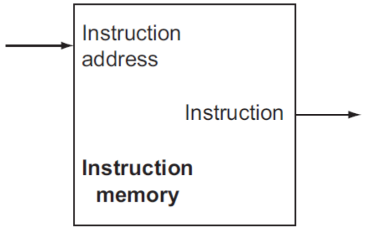
Building a Datapath: State Elements



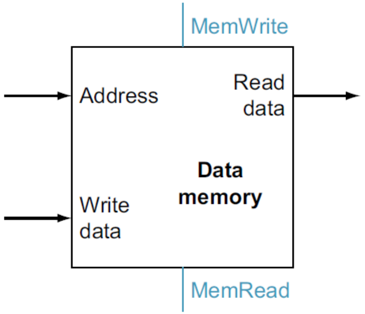
- The register file always outputs the contents of whatever register numbers are on the Read register inputs.
- Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge.

Building a Datapath: State Elements

- In our processor, we'll use two "separate" memories:
  1. **IMEM**: A read-only memory for fetching instructions.
  2. **DMEM**: A memory for loading (read) and storing (write) data words.

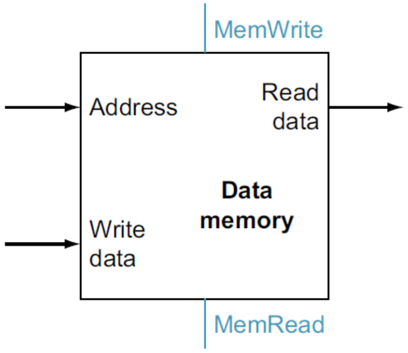


Instruction memory



Data memory unit

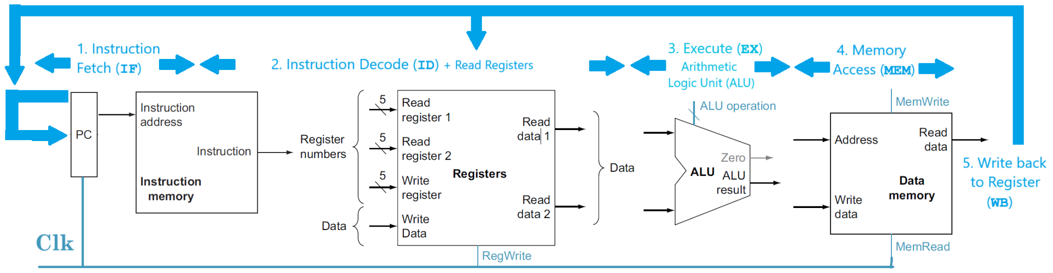
Building a Datapath: State Elements



Data memory unit

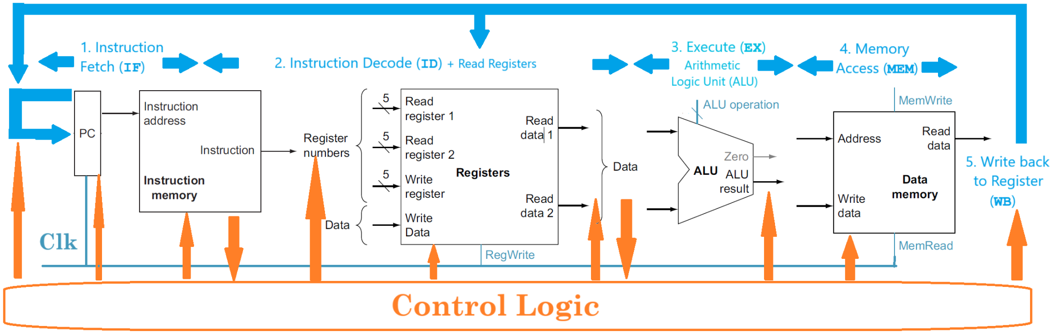
- Memory words are accessed as follows:
  - **Read**: Set MemRead=1.
    Address selects word to put on Read data bus.
  - **Write**: Set MemWrite=1.
    Address selects word to be written with Write data bus.
- Like Register File, clock input is only a factor on write (write occurs on rising clock edge).

Five Basic Stages (Phases) of Instruction Execution



Five Basic Stages (Phases) of Instruction Execution

- Not All Instructions Need All five Stages!
- The control logic selects “needed” datapath lines based on the instruction.



Building a datapath

A Basic RISC-V Implementation

We will be examining an implementation that includes a subset of the core RISC-V instruction set:

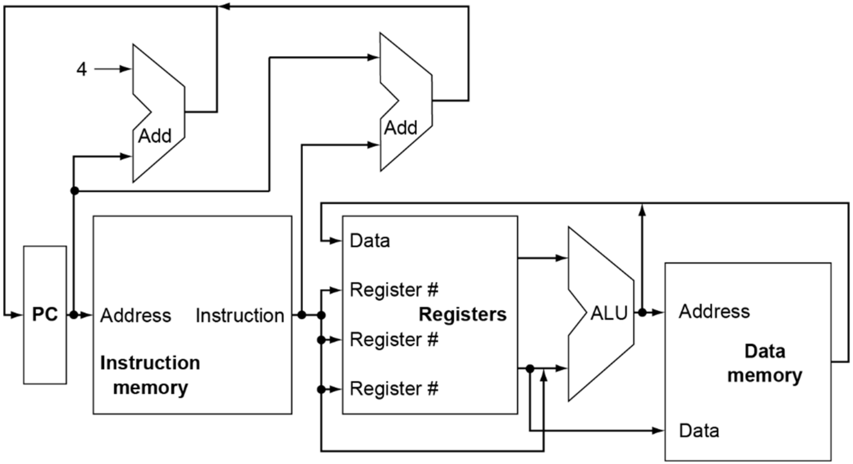
- The memory-reference instructions load word (lw) and store word (sw).
- The arithmetic-logical instructions add, sub.
- The conditional branch instruction branch if equal (beq).

An Overview of the Implementation

For every instruction, the first two steps are identical:

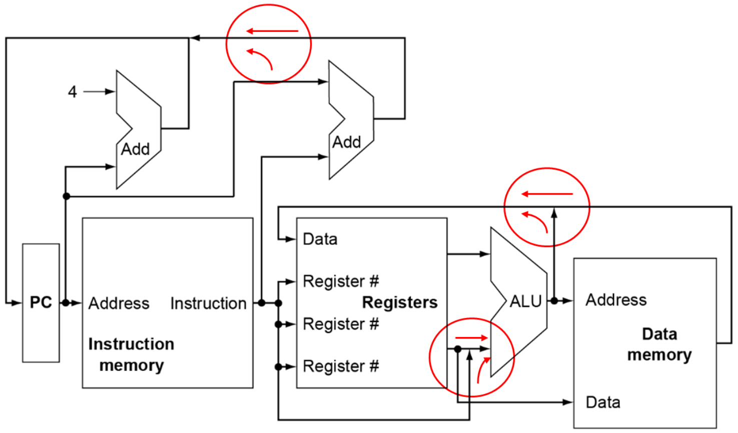
1. Send the **program counter (PC)** to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the lw instruction, we need to read only one register, but most other instructions require reading two registers.
3. After these two steps, the actions required to complete the instruction depend on the instruction class.

An Overview of the Implementation



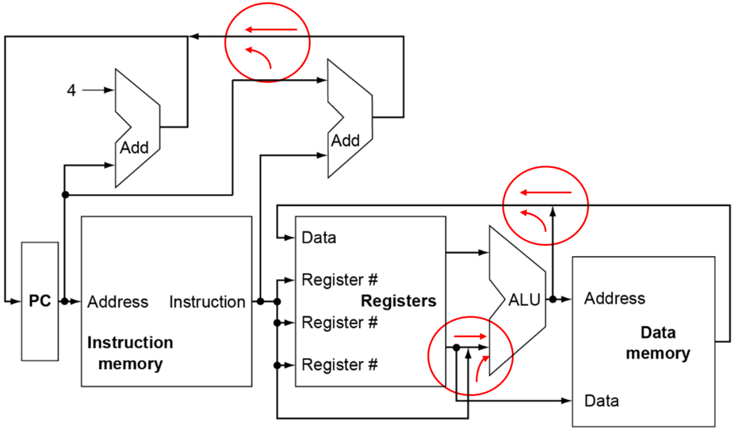
•

An Overview of the Implementation



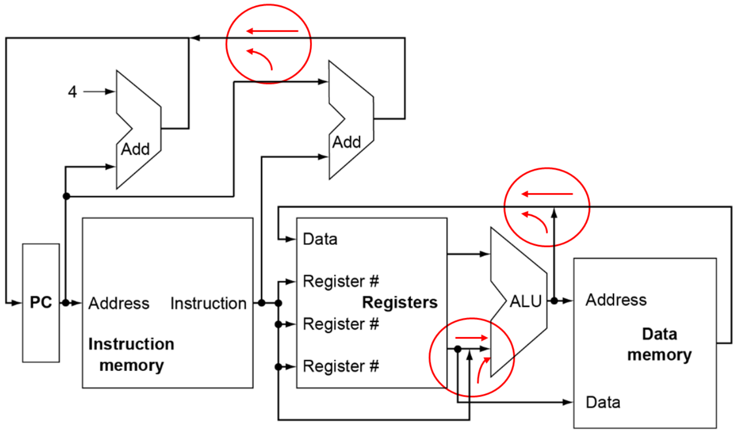
- Can’t just join wires together
- Use multiplexers

An Overview of the Implementation



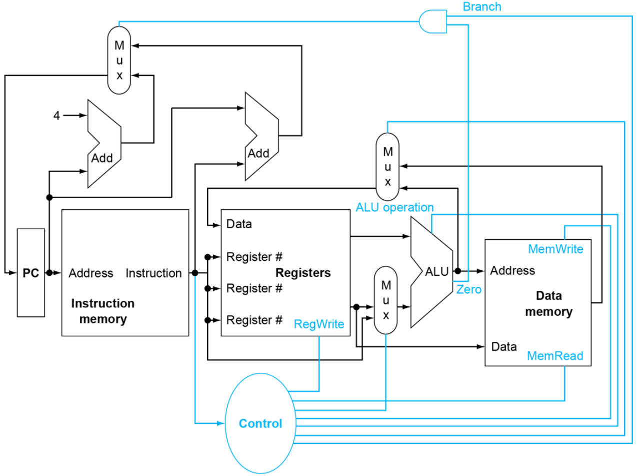
- Several of the units must be controlled depending on the type of instruction.
- **For example**, the data memory must read on a load and write on a store.

An Overview of the Implementation



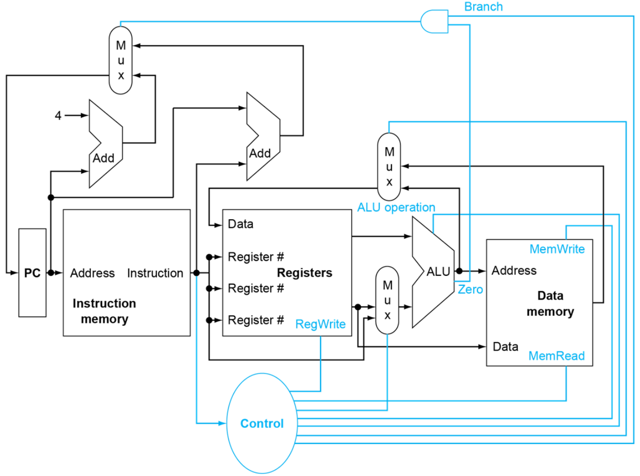
- The register file must be written only on a load or an arithmetic-logical instruction.
- And, of course, the ALU must perform one of several operations.
- **We need some Control!!**

Control



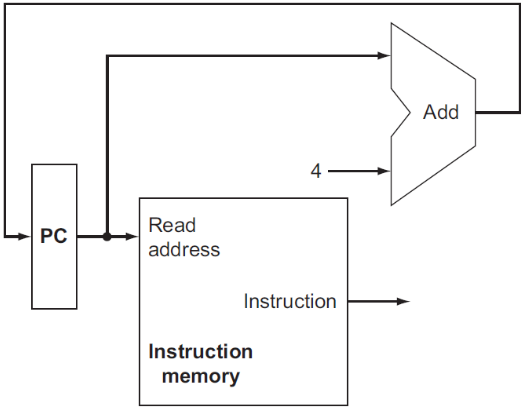
- A **control unit**, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors.

Control



- The top multiplexor, which determines whether **PC + 4** or the **branch destination address** is written into the **PC**, is set based on the **Zero** output of the ALU, which is used to perform the comparison of a **beq** instruction.

Building a Datapath

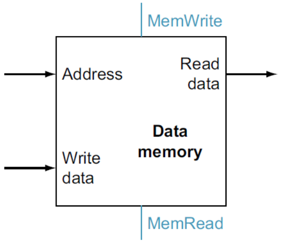


- To execute any instruction, we must start by fetching the instruction from memory.
- To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later.

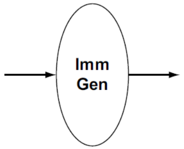
Building a Datapath (Load and Store)

- These instructions compute a memory address by adding the base register to the 12-bit signed offset field contained in the instruction.
- If the instruction is a store, the value to be stored must also be read from the register file where it resides.
- If the instruction is a load, the value read from memory must be written into the register file in the specified register.
- Thus, we will need both the register file and the ALU.

Building a Datapath (Load and Store)



a. Data memory unit



b. Immediate generation unit

- Furthermore, we will need a unit to sign-extend the 12-bit offset field in the instruction to a 32-bit signed value.
- A data memory unit to read from or write to.
- The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory.

Building a Datapath (Load)

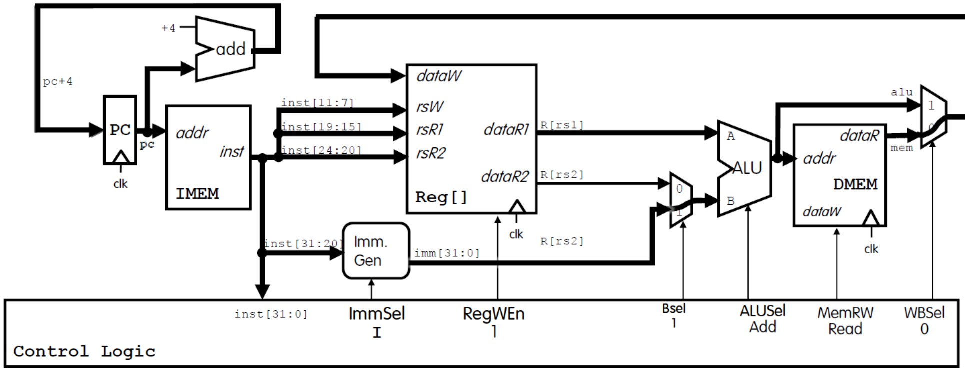
Increment PC to next instruction.

Immediate Generation Block builds a 32-bit immediate **imm**.

ALU computes address **alu = R[rs1] + imm**.

Read memory at address **alu**.

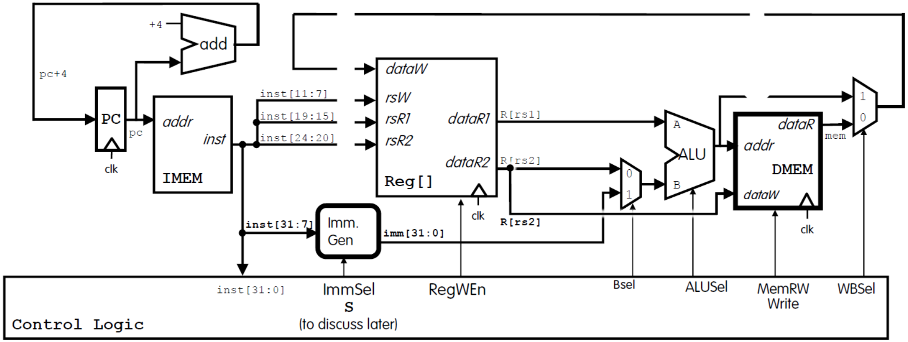
Write loaded memory value **mem** to register.



Building a Datapath (Store)

Control ImmSel selects how to generate immediate type: `I`, `S`.

Control MemRW=Write saves `rs2` to memory on the next rising clock edge.



Building a Datapath (beq)

- The beq instruction has three operands, two registers that are compared for equality, and a 12-bit offset used to compute the branch target address relative to the branch instruction address.
- **Branch target:** *address The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the RISC-V architecture, the branch target is given by the sum of the offset field of the instruction and the address of the branch*.

Building a Datapath (beq)

- There are two details in the definition of branch instructions to which we must pay attention:
  1. The instruction set architecture specifies that the base for the branch address calculation is the address of the branch instruction.
  2. The architecture also states that the offset field is shifted left 1 bit so that it is a half word offset; this shift increases the effective range of the offset field by a factor of 2.
- To deal with the latter complication, we will need to shift the offset field by 1.

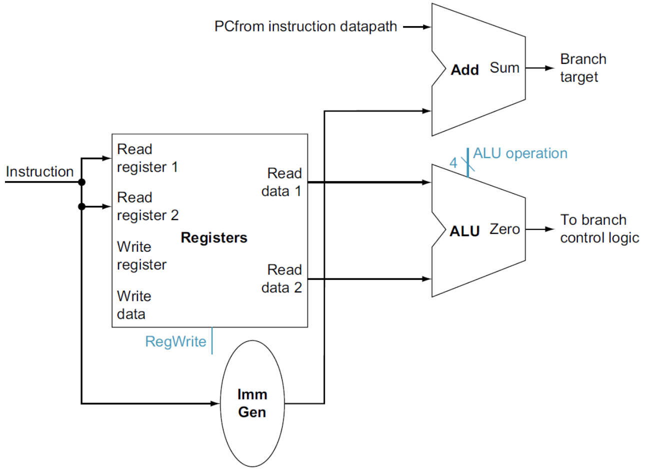
Building a Datapath (beq)

- As well as computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address.
- When the condition is true (i.e., two operands are equal), the branch target address becomes the new PC, and we say that the **branch is taken**.
- **Branch taken:** *A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches*.

Building a Datapath (beq)

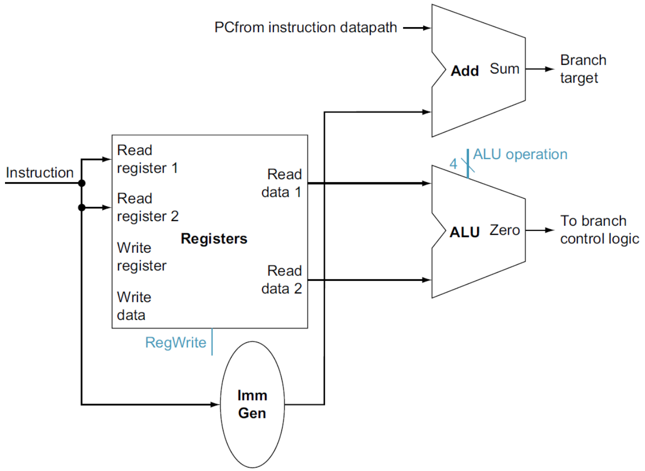
- If the operand is not zero, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the **branch is not taken**.
- **branch not taken or (untaken branch)** *A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch*.

Building a Datapath (beq)



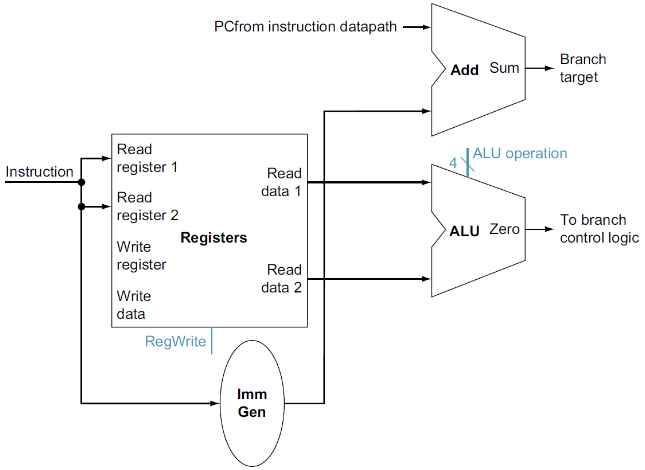
- To compute the branch target address, the branch datapath includes an immediate generation unit and an adder.
- To perform the compare, we need to use the register file to supply two register operands.

Building a Datapath (beq)



- In addition, the equality comparison can be done using the ALU.
- Since that ALU provides an output signal that indicates whether the result was 0, we can send both register operands to the ALU with the control set to subtract two values.

Building a Datapath (beq)



- If the Zero signal out of the ALU unit is asserted, we know that the register values are equal.
- The branch instruction operates by adding the PC with the 12 bits of the instruction shifted left by 1 bit.
- Simply concatenating 0 to the branch offset accomplishes this shift.

Building a Datapath (Branch)

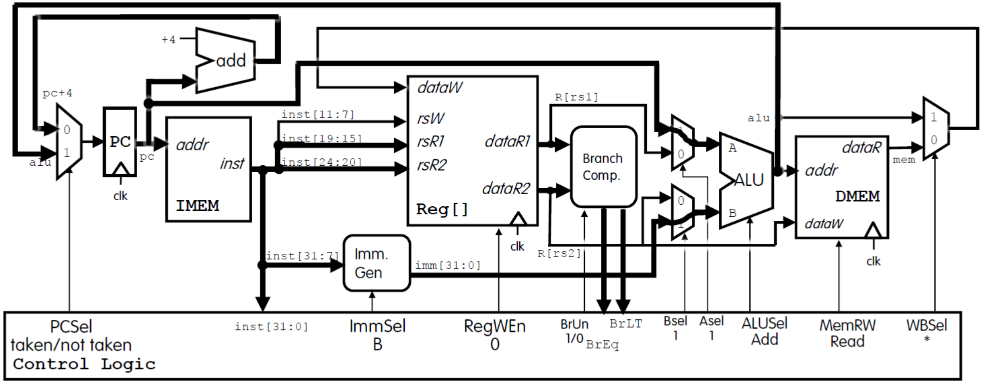
If PCSel=taken, update PC to ALU output. Else, update to next instruction **PC + 4**.

Build **imm** from B-type instruction.

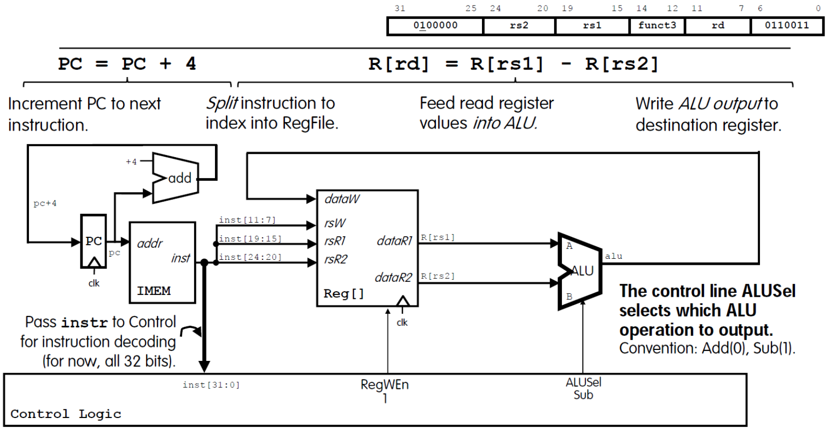
- Compute branch; feed to Control.
- Compute **PC + imm**.

Don't write to memory.

Don't write to registers.



Building a Datapath (add/sub)



PC = PC + 4

R[rd] = R[rs1] - R[rs2]

Increment PC to next instruction.

Split instruction to index into RegFile.

Feed read register values *into ALU*.

Write *ALU output* to destination register.

Pass **instr** to Control for instruction decoding (for now, all 32 bits).

The control line ALUSel selects which ALU operation to output.
Convention: Add(0), Sub(1).

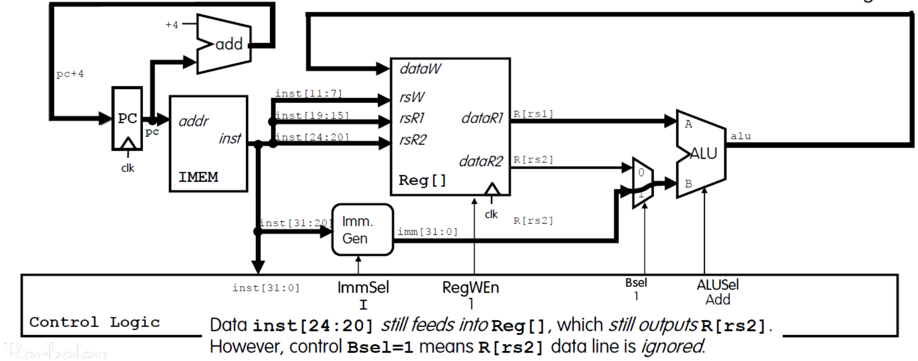
Building a Datapath (addi)

Increment PC to next instruction.

Immediate Generation Block builds a 32-bit immediate **imm** from instruction bits.

Control line Bsel=1 selects the generated immediate **imm** for ALU input B.

Write ALU output to destination register.



Data **inst[24:20]** *still feeds into* **Reg[]**, which *still outputs* **R[rs2]**.
However, control **Bsel=1** means **R[rs2]** data line is *ignored*.

Building a Datapath (jal)

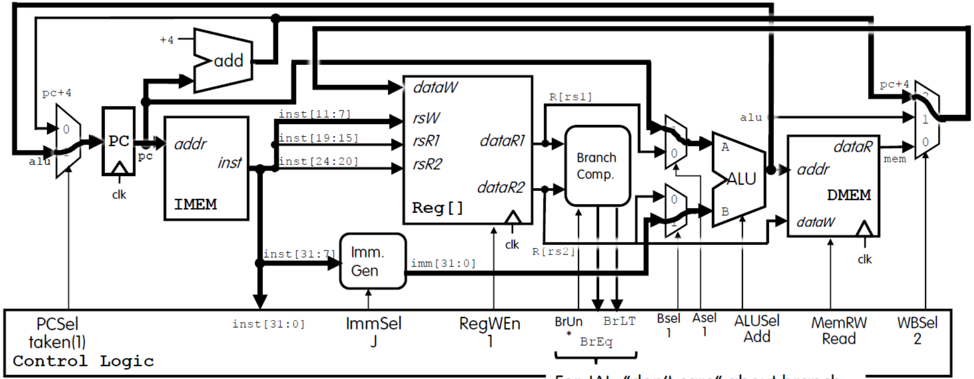
- Feed PC into blocks.
- Write ALU output to PC.

Generate byte offset **imm** for 20-bit PC-relative jump.

Compute **PC + imm**.

Write **PC + 4** to destination register.

Don't write to memory.



For JAL, "don't care" about branch.

Building a Datapath (jalr)

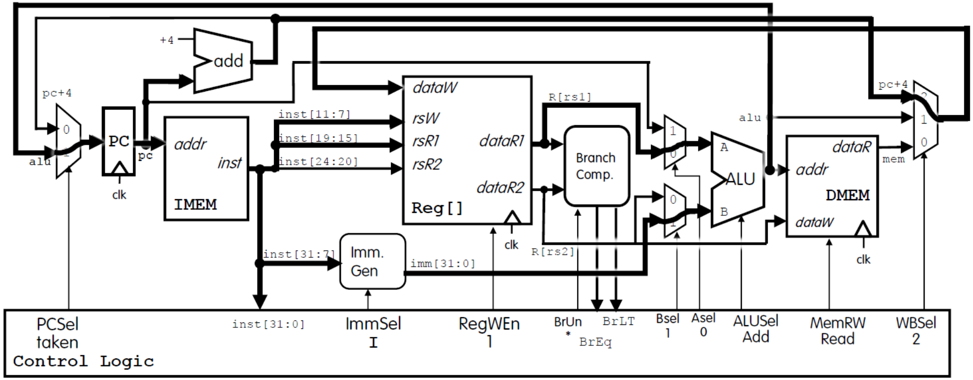
- Feed PC into blocks.
- Write ALU output to PC.

Generate 12-bit **imm** (**I**-Format).

Compute **rs1 + imm**.

Write **PC + 4** to destination register.

Don't write to memory.



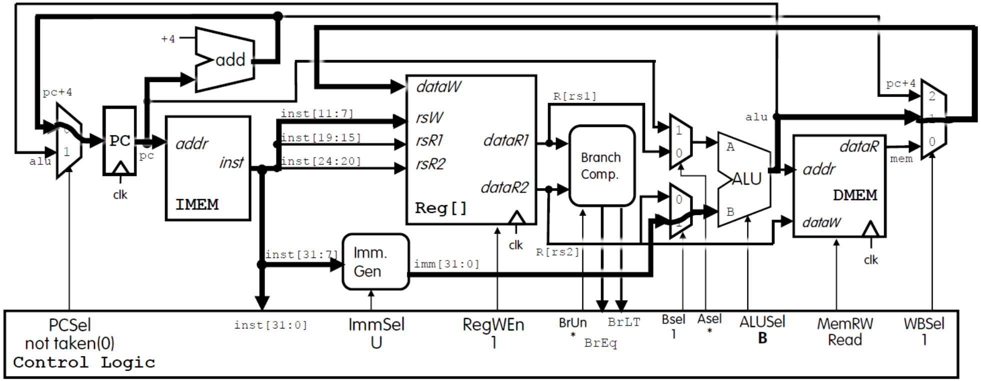
Building a Datapath (lui)

Increment PC to next instruction.

Generate `imm` with upper 20 bits. (U-format)

Grab only `imm`! (ALUSel=B)

Write result to destination register. Don't write to memory.



Building a Datapath (auipc)

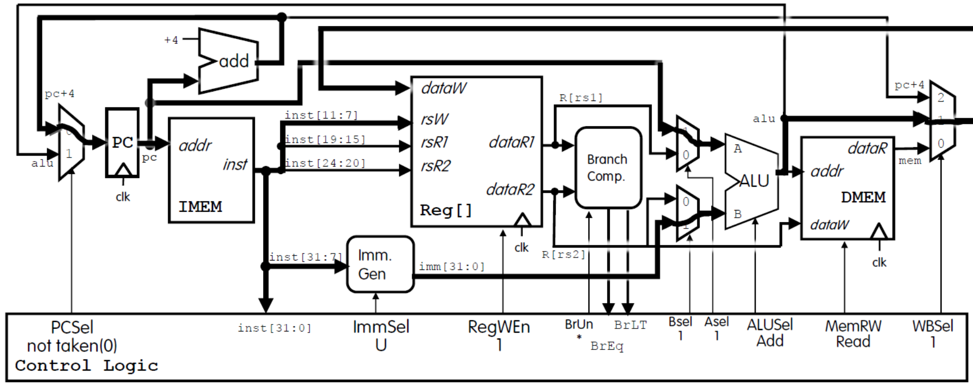
Increment PC to next instruction.

Generate `imm` with upper 20 bits. (U-format)

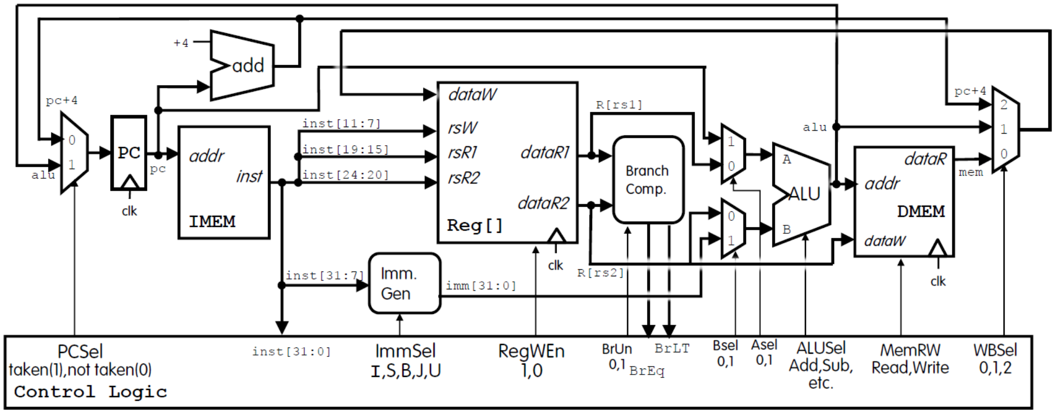
Add `PC + imm`!

Write result to destination register.

Don't write to memory.



Complete RV31I Datapath



Why a Single-Cycle Implementation is not Used Today

- Notice that the clock cycle must have the same length for every instruction in this single-cycle design.
- The longest possible path in the processor determines the clock cycle (load instruction).
- The overall performance of a single-cycle implementation is likely to be poor since the clock cycle is too long.
- Historically, early computers with very simple instruction sets did use this implementation technique.

ensia

Thank you!