Hardware overview

Three basic classes of instructions in most ISAs

Computations

Add two variables together, subtract a constant, perform bit operations...

Data transfer

Loads and stores from memory

Control logic

Conditional and unconditional jumps (support for if-statements, loops, function calls)

Data needs to be easily accessible by the CPU in order to do this

Arithmetic and logical operations

RISC-V (32I base instruction set) offers the following:

- Arithmetic: ADD, SUB
- Logical: AND, OR, XOR
- Shift: SLL, SRL, SRA
- Comparison: SLT[U]

No need for ADDU because 2s complement math "just works" (see textbook)
Compiler takes care of applying SRL to unsigned vars and SRA to signed

Three operands: two source registers (*rs1*, rs2), one destination (**rd**)

```
sub x12, x11, x10 # x12 = x11 - x10
or x13, x13, x14 # x13 |= x14
slt x17, x15, x16 # x17 = x15 < x16 ? 1 : 0
```

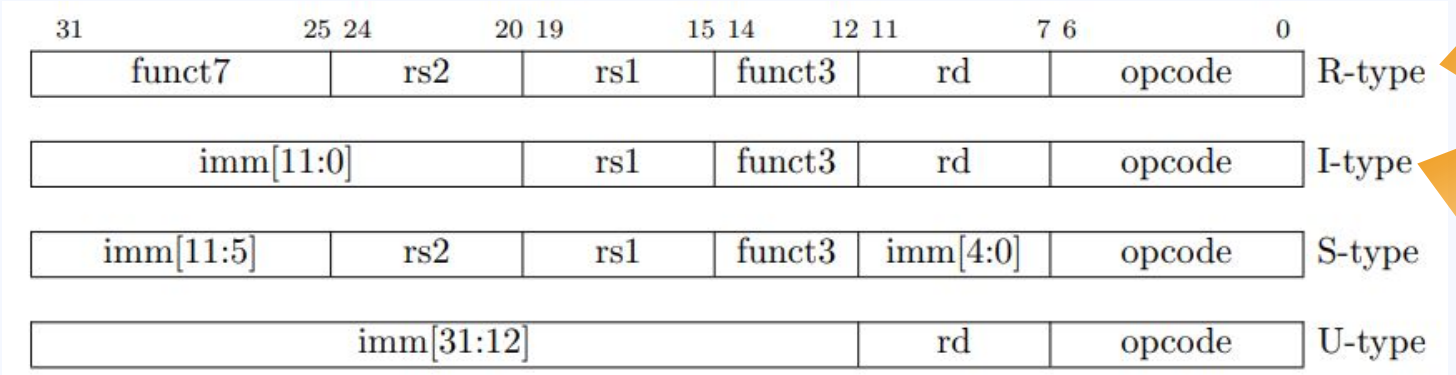
If an arithmetic/logical instruction has three operands, how do you add a constant to a register?

Immediates + instruction formats

Immediates: constants that are encoded as part of an instruction

Separate instructions from register-register (ADD vs ADDI)

In RISC-V, register-immediate instructions have rd, rs1, and immediate operand



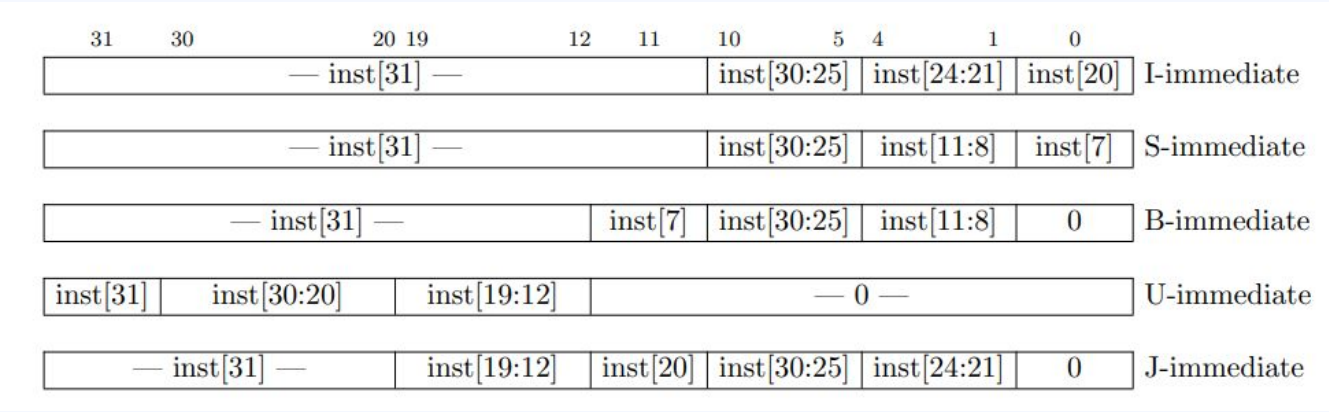
Register-register and Register-immediate instrs

Why no SUBI?

Arithmetic	ADD, SUB	ADDI
Logical	AND, OR, XOR	ANDI, ORI, XORI
Shift	SLL, SRL, SRA	SLLI, SRLI, SRAI
Comparison	SLT[U]	SLTI[U]

```
addi x10, x10, 2 # x10 += 2
xori x12, x11, 0xff # x12 = x11 ^ 0xff
slli x14, x13, 16 # x14 = x13 << 16
```

Immediate encodings



only 12 bits of an I-type instruction are set aside for the immediate

value is *sign-extended* into 32 bits

- need to pay attention to MSB (xori x10, x10, 0xfff will flip all bits, not just the last 12)
- loading a large number into a register may take multiple instructions (hint: LUI)

source // another visualization // Babbage's analytical engine

? ? ?

Goal of a CPU

Interpret an instruction (bits in memory/signals on wires) as an action it should take

What does that look like in hardware?

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB

HW assumptions we're working with

CPU can read bits from memory as electrical signals (one "wire" per bit)

Everything we read is a pure low/high signal, no noise/interference

Not worried about constraints (space, complexity, power) **for now**

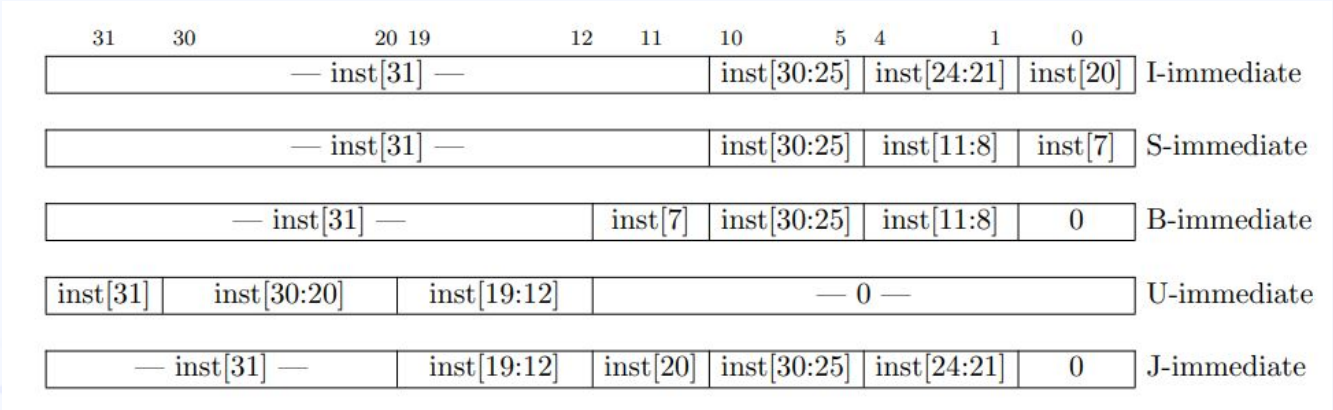
Not assuming that signals stabilize instantaneously

To run a program, CPU HW needs:

A way to extract/rearrange bits - pull out the relevant fields of an instruction

A way to implement combinational logic – arithmetic/logical, comparison

A way to keep track of state - what is the value of the PC at the current step?



Data as collections of wires

wire/data line: carries a single digital signal (on/off)

bus (P&H definition): a collection of data lines that is treated as one, multi-bit signal

Combinational logic circuits

Examples: adders, logical operators, control signal translation

Work like pure functions (no memory)

Combinatorial expressions can be automatically synthesized to circuits

Physically, logic gates are implemented using transistors (electrical switches)

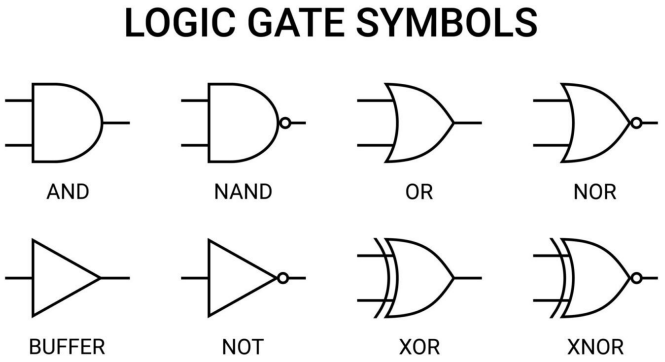


image source

"half adder" (**s**um and **c**arry output):

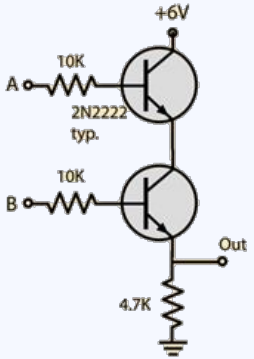


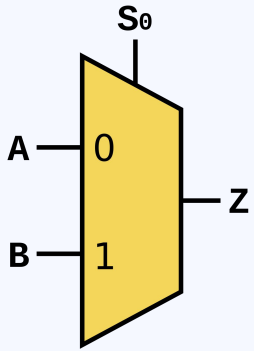
image source

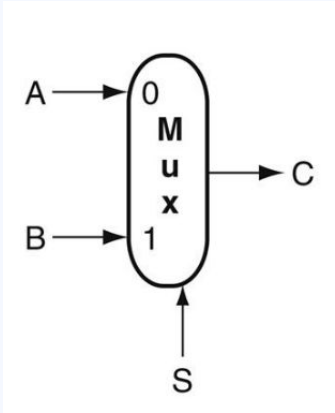
Multiplexers

Used to select between multiple inputs

n-bit selector signal = select between 2^n inputs

Example: 2nd operand for add vs addi

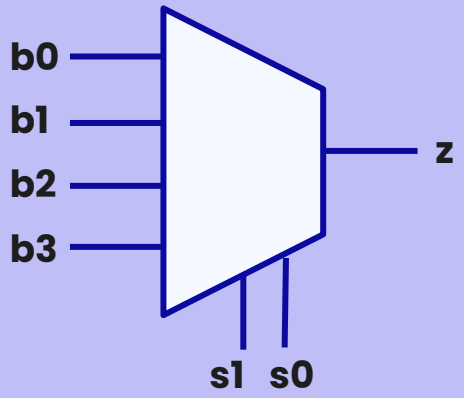


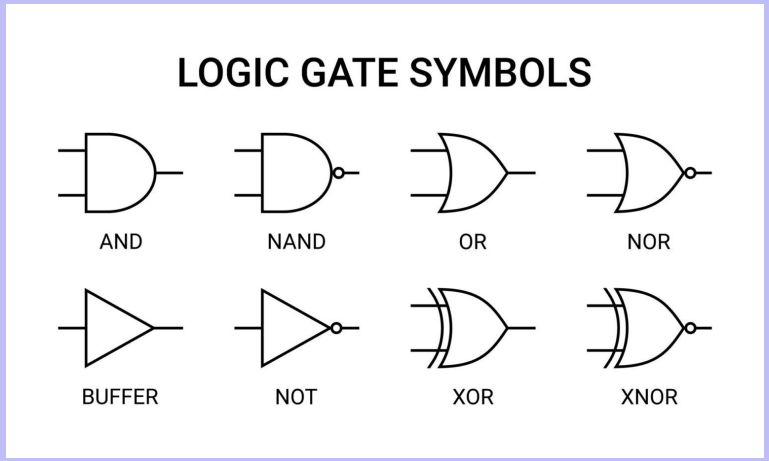


P&H Fig. A.3.2

? ? ?

Build a 4-input (2-bit selector) mux out of logic gates

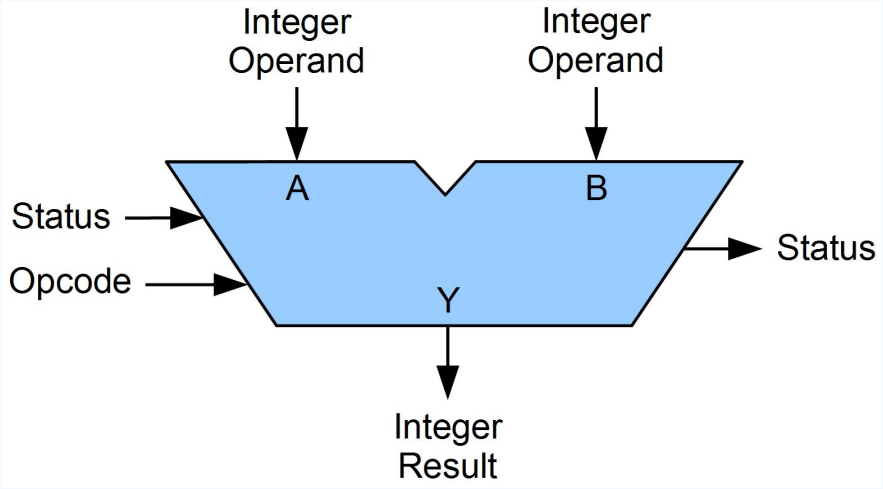




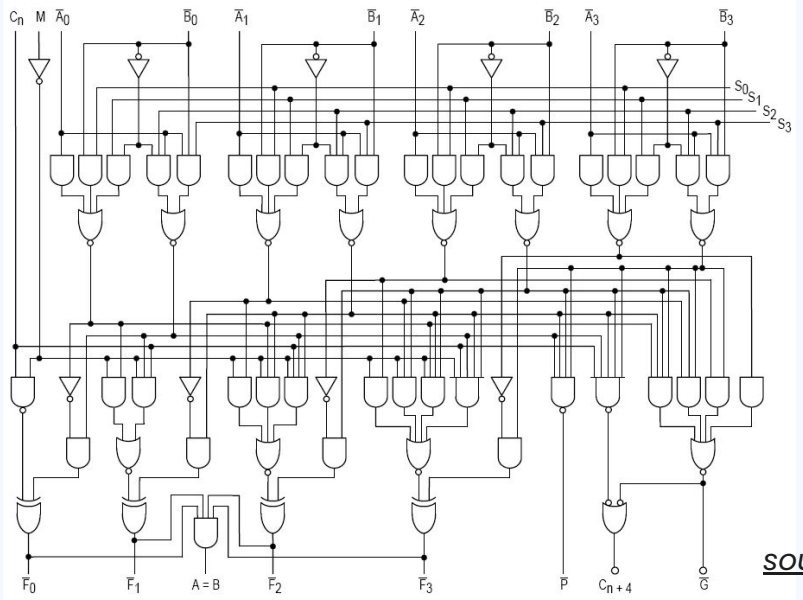
Arithmetic Logic Unit

"ALU"

Takes in two operands and a control signal for the operation, produces result of applying operation on operands (status input/output signals optional)



By Lambtron - Own work, CC BY-SA 4.0, link



source

Clocks

In a circuit, many things happen in parallel

Synchronization signal (wire) that allows necessary components to know when to move on to the next “step”

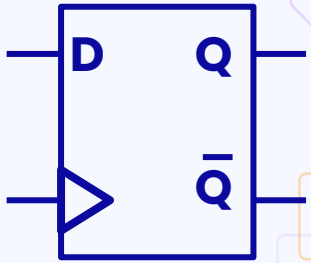
Clock cycle time is long enough to allow for signals to **stabilize**

> i.e. allow electrons to travel through the longest possible path of wires/transistors

Components that have state

How do we express “at each step, increment the PC by 4?”

Memory elements, such as flip flops and latches, have internal state that updates on clock tick (D flip-flop pictured)



Clocked register: a multi-bit memory element that updates on clock tick

How do we express “at each clock tick, increment the PC by 4” using a clocked register (for the PC) and an adder?

Takeaways

Bits of an instruction = electrical signals CPU uses to execute a program

CPU is just a (very, very) big circuit made up of wires, combinational logic elements, and memory elements

Can implement modules we need (multiplexers, ALUs, bit selectors, registers) using these elements

Basically: we have an “existence proof” of the hardware we need, so we can start working one level of abstraction higher to implement a CPU

Let’s build a CPU!

What do we need to get started?