Assembly and processors

Learning objectives

- Don’t be scared of assembly code
  - understand what it’s for and why
  - know the pieces used in all assemblies
- Outline the design of computer processors
- Use terminology without embarrassment:
  - source code, assembly, machine code
  - compiler, assembler, linker, loader

Let’s try to make this code use less complicated individual steps (but more of them).

```
int i;
for(i = 0; i < n; i++) {
    data[i] = (i * (i+1)) / 2;
}
printf("done!\n");
```

Change the loop to a simpler form

```
int i;
i = 0;
while(i < n) {
    data[i] = (i * (i+1)) / 2;
    i++;
}
printf("done!\n");
```

Split the math to be one operation per statement

```
int i, tmp;
i = 0;
while(i < n) {
    tmp = i + 1;
    tmp = i * tmp;
    tmp = tmp / 2;
    data[i] = tmp;
    i++;
}
printf("done!\n");
```

Convert the array notation to pointer notation

```
int i, tmp; void *ptr;
i = 0;
while(i < n) {
    tmp = i + 1;
    tmp = i * tmp;
    tmp = tmp / 2;
    ptr = i * sizeof(int);
    ptr = ptr + data;
    *(int *)ptr = tmp;
    i++;
}
printf("done!\n");
```

Remove the `++` and `sizeof` shorthand.

```
int i, tmp; void *ptr;
i = 0;
while(i < n) {
    tmp = i + 1;
    tmp = i * tmp;
    tmp = tmp / 2;
    ptr = i * 4;
    ptr = ptr + data;
    *(int *)ptr = tmp;
    i = i + 1;
}
printf("done!\n");
```

Change the loop into explicit moves

```
1  int i, tmp; void *ptr;
2  i = 0;
3  if (i >= n) goto line 12;
4  tmp = i + 1;
5  tmp = i * tmp;
6  tmp = tmp / 2;
7  ptr = i * 4;
8  ptr = ptr + data;
9  *(int *)ptr = tmp;
10 i = i + 1;
11 goto line 3;
12 printf("done!\n");
```

Move comparison and string assignment to their own lines

```
1  int i, tmp, ok; void *ptr, *s;
2  i = 0;
3  ok = i >= n;
4  if (ok) goto line 13;
5  tmp = i + 1;
6  tmp = i * tmp;
7  tmp = tmp / 2;
8  ptr = i * 4;
9  ptr = ptr + data;
10 *(int *)ptr = tmp;
11 i = i + 1;
12 goto line 3;
13 s = "done!\n";
14 printf(s);
```

Replace variables with predetermined set of “program registers.” The arguments (`n` and `data`) get the first two (`r0` and `r1`), then locals in the order they are used.

```
1  r2 = 0;
2  r3 = r2 >= r0;
3  if (r3) goto line 12;
4  r4 = r2 + 1;
5  r4 = r2 * r4;
6  r4 = r4 / 2;
7  r5 = r2 * 4;
8  r5 = r5 + r1;
9  *(int *)r6 = r4;
10 r2 = r2 + 1;
11 goto line 2;
12 r6 = "done!\n";
13 printf(r6);
```

Function calls are two parts: copying the parameters into expected program registers; then a special kind of `goto` we can return from using the stack.

```
r2 = 0;
r3 = r2 >= r4;
if (r3) goto line 12;
r5 = r2 + 1;
r5 = r2 * r5;
r5 = r5 / 2;
r6 = r2 * 4;
r6 = r6 + r0;
*(int *)r6 = r5;
r2 = r2 + 1;
goto line 2;
r7 = "done!\n";
r0 = r7;
call printf;
```

What we’re left with

- arithmetic/logic operations:
  - variable = constant
  - variable = variable *op* variable
- memory operations:
  - variable = *pointer
  - *pointer = variable
- instruction sequence control operations:
  - if variable, go to code location
  - go to code location
  - call
  - return

Assembly (sometimes called assembler code):

- Simple line-oriented textual encoding
- Format `operation operand, operand`

```
mov     rax, 1
xor     rdi, rdi
cmp     r9, r8
```

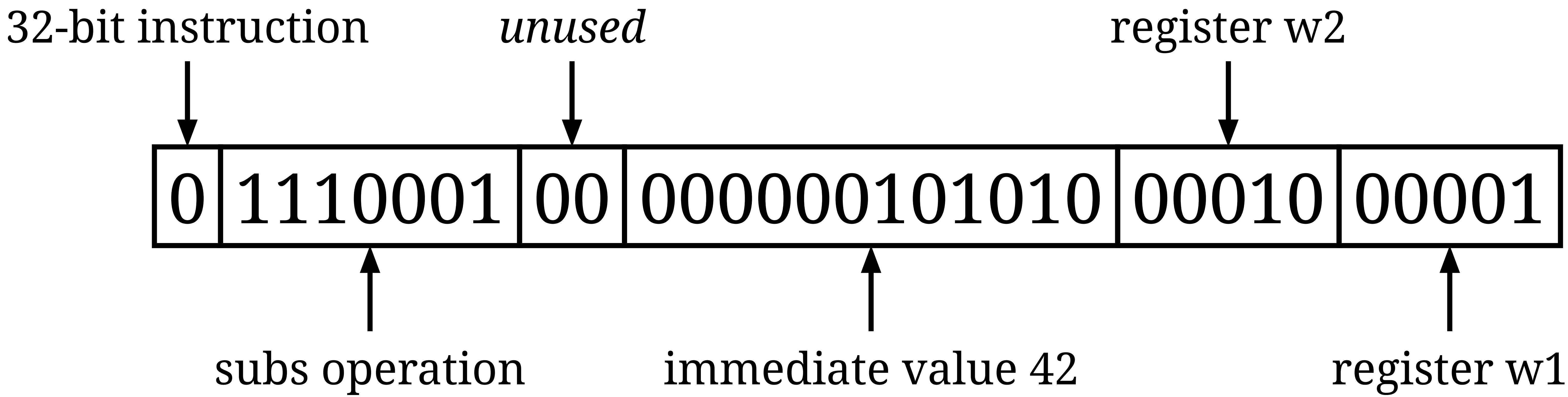
- Code locations abstracted by labels

```
somename:
    inc     rdx
    cmp     rdx, r9
    jng     somename
```

- Set of operations, registers, and memory addressing syntax vary by target hardware

Machine code is a binary encoding of operations

Single instruction example from aarch64:



In assembly: `subs w1, w2, 42`

In C: `w1 = w2 - 42;` and compares the result to 0 for future conditional jumps

Multiple instructions are concatenated in memory

Vocabulary

Term	Meaning
Instruction	Single action sent to processor
Machine code	Binary representation of individual instructions
Assembly	Textual representation of individual instructions (with labels instead of raw addresses for jumps)
Source code	Code in a “high-level” programming language (not assembly; this includes all code you’ve written)
Instruction set architecture (ISA)	Computer design at the level of what machine code they understand
Jump	An instruction that picks a different (not next-in-memory) instruction to run next

Source code (`.c`, `.java`, `.py`, etc)

1. **Compile** to assembly (most compilers also assemble and link)

Assembly code (`.s`, `.S`)

2. **Assemble** to object files

Object files, both static (`.o`, `.obj`, `.a`) and shared (`.so`, `.dylib`, `.dll`)

3. **Link** static files into an executable (shared files get linked during loading)

Executable files (no extension, or `.exe`)

4. **Load** into memory (both executable and shared object files)

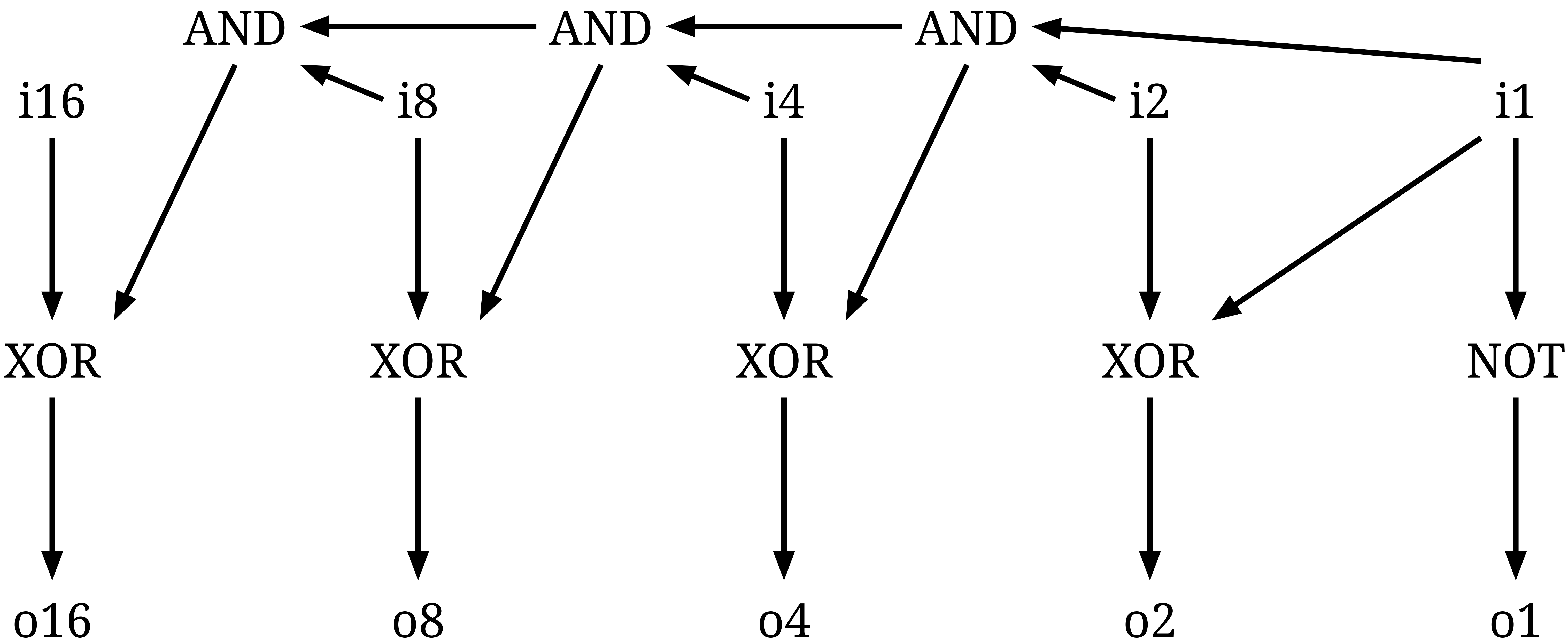
Machine code in memory

5. **Execute** by running the instruction in the first byte of the program

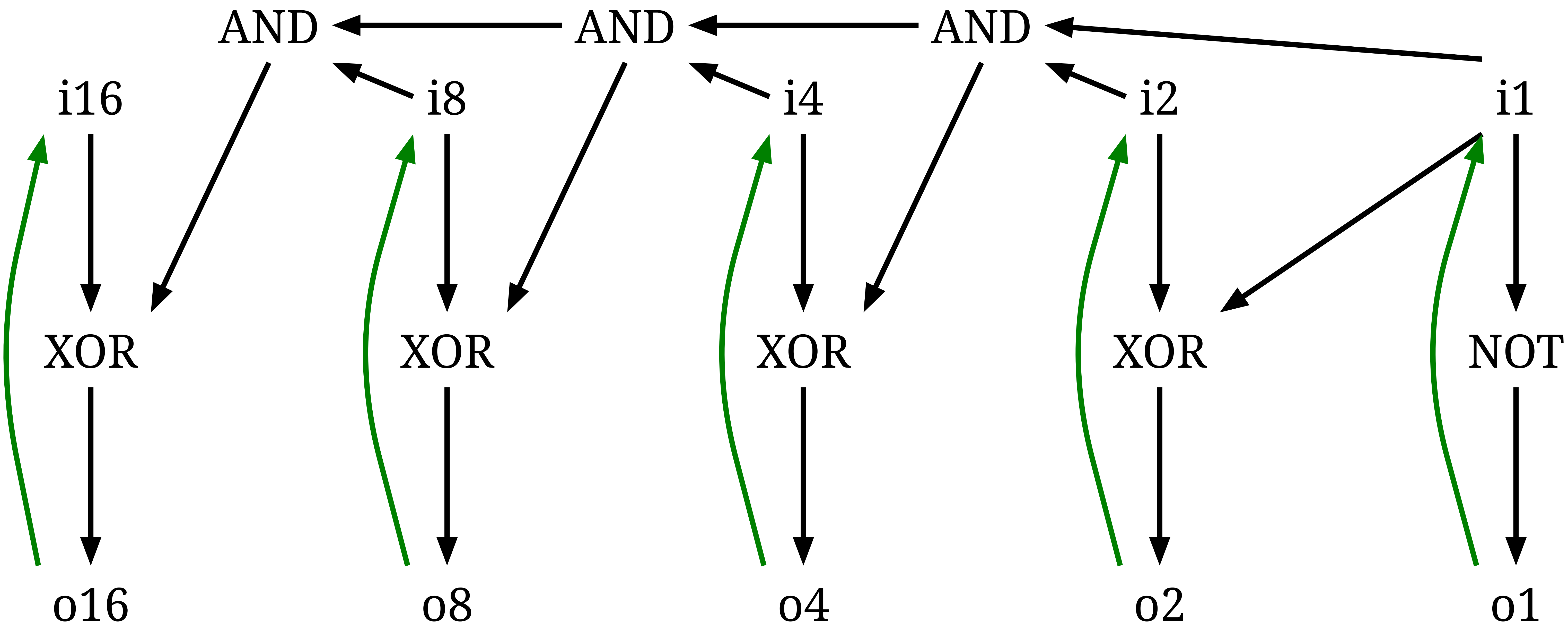
Building a Processor

- von Neumann architecture:
  - Memory = one big array of bytes (both code and data)
- Registers
  - High-speed on-processor memory
    - (Built using six NAND gates per bit)
  - Few in number; usually 32 or 64 bits in size each
  - Clocked
    - Usually: register outputs its stored value; input is ignored
    - When clock bit changes, input copied into stored value

What does this ciruit do?

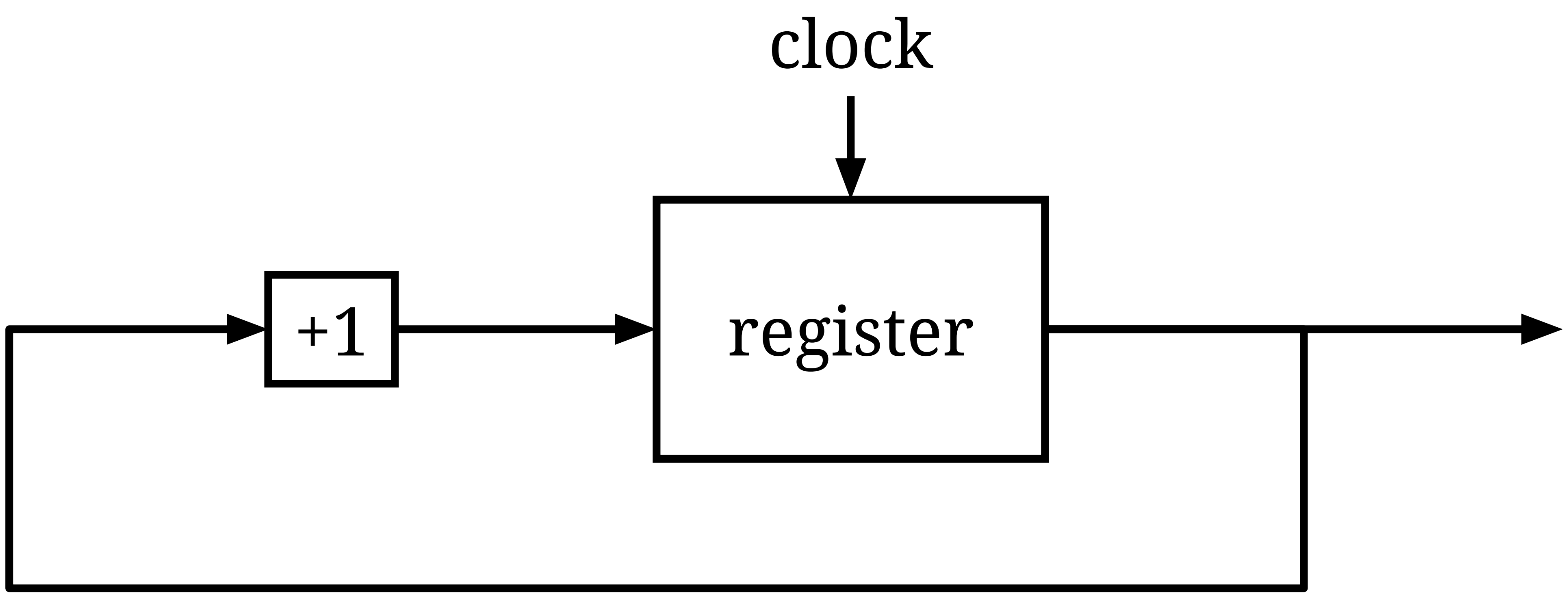


What does this ciruit do over time?



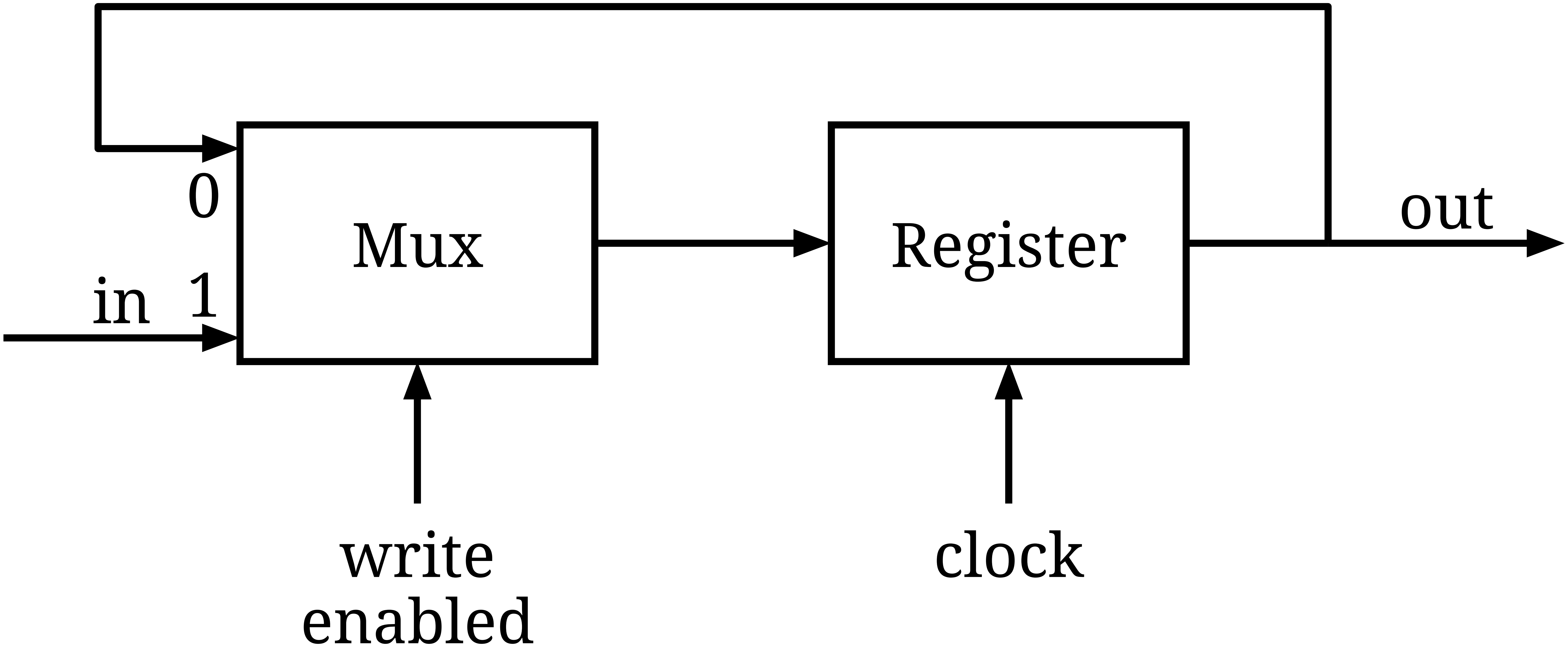
Need for registers

- Logic is made of many gates
- Gates take time to settle
- Registers wait for all gates to settle (using a clock)



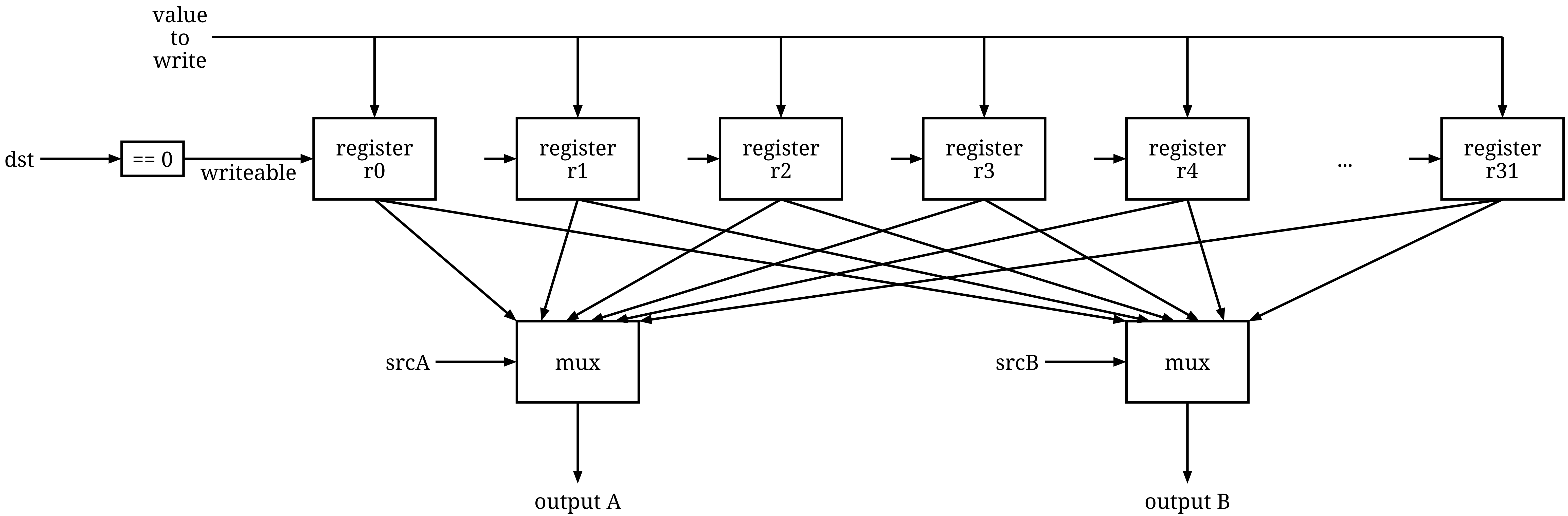
- **Frequency scaling** slows down clock when not much going on (saves power)
- **Overclocking** uses faster clock than chip designers think is safe

Selectively-writeable register



Register file

- Goal: support things like `r8 = r9 + r10`: up to 2 reads and 1 write, selected by index



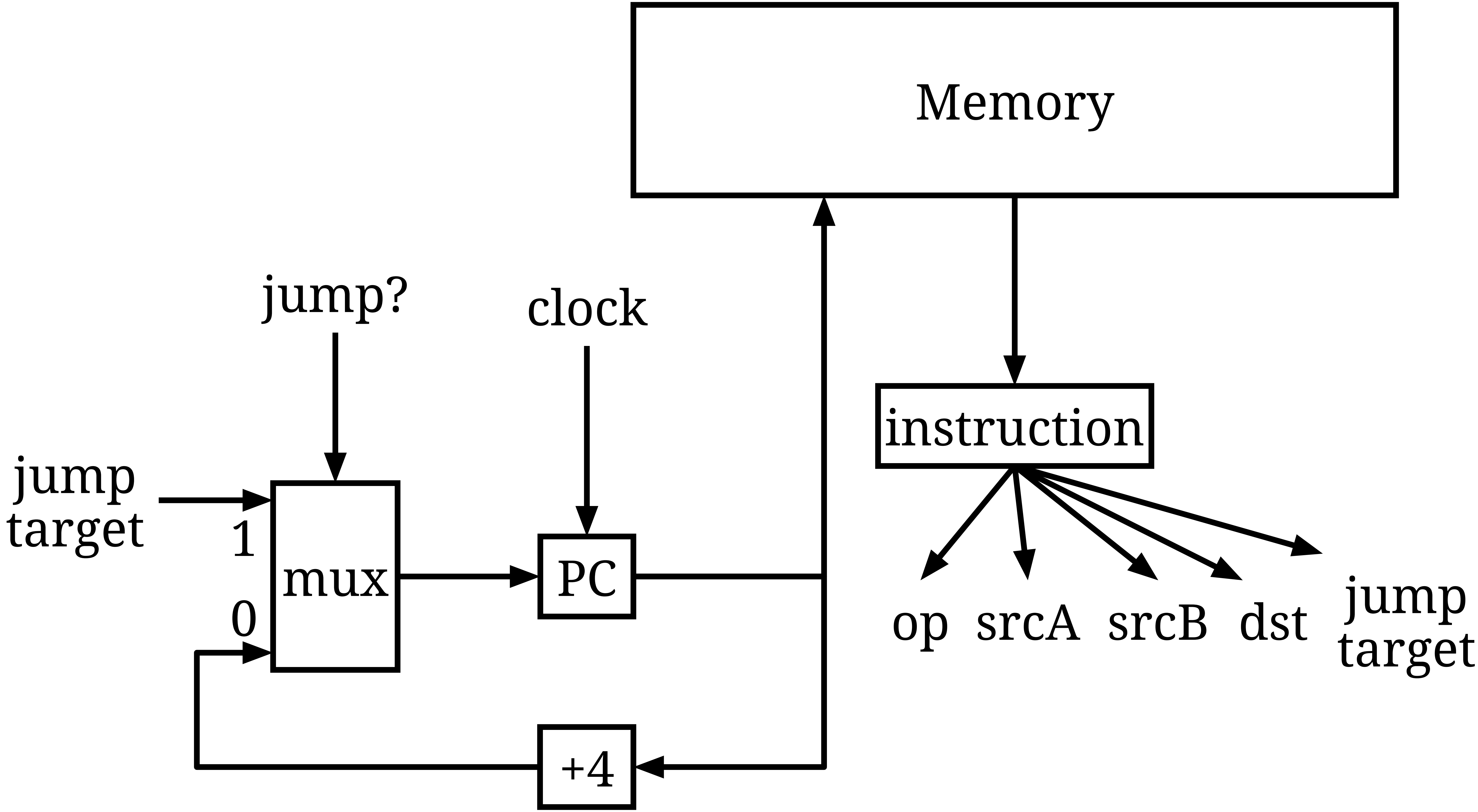
Arithmetic Logic Unit (ALU)

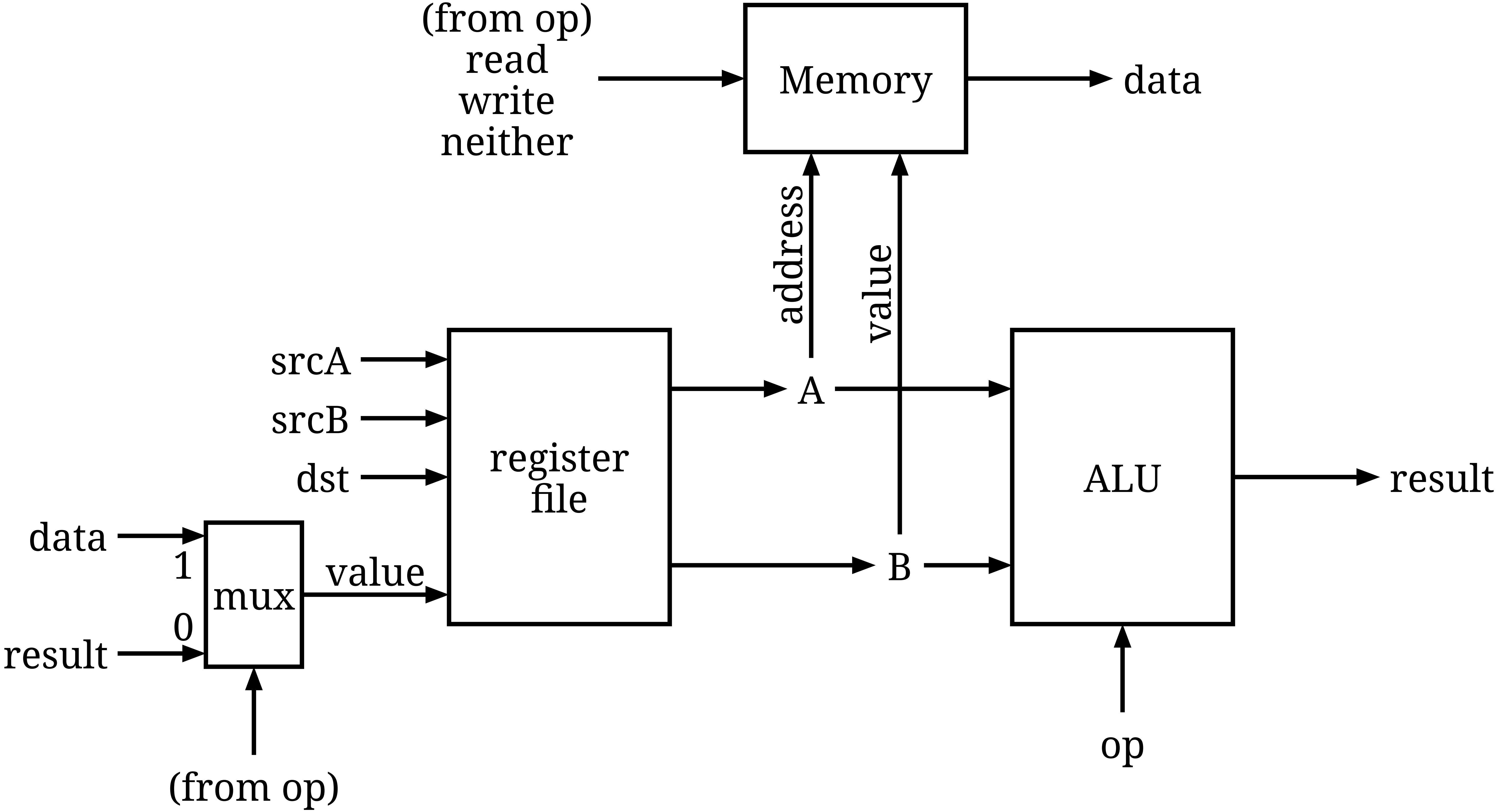
- Goal: pick what operation to perform
- Simple version:
  - Build a `+` circuit
  - Build a `-` circuit
  - Build a `*` circuit
  - Build a `/` circuit
  - Build a `<` circuit
  - Build a `&` circuit
  - …
  - Send operands to *all* of those circuits
  - Pick *one* circuit’s output with a Mux
- Fancier versions save power by not sending operands to unused circuits

Putting it together

1. A register called the *program counter* (PC) stores the next address to *fetch* an instruction from
2. The instruction is loaded from memory at that address
3. The instruction is *decoded*, broken into pieces with
   - operation sent to the ALU
   - srcA, srcB, and dst sent to the register file
   - jump target sent to the net PC stage
   - memory address sent to memory
4. The instruction is *executed*, letting the ALU and memory do their thing
5. The processor *writes back* the results, meaning:
   - if the instruction had a register destination, the register file updates
   - if the instruction was a jump, the target is written to the PC

     otherwise, the old PC value + the size of the instruction is written to the PC
6. The clock ticks and the whole thing repeats for the next instruction


Memory
jump?
clock
instruction
jump
target
1
mux
0
PC
+4
op
srcA
srcB
dst
jump
target


(from op)
read
write
neither
Memory
data
address
value
srcA
srcB
dst
register
file
A
B
ALU
result
data
1
mux
0
result
value
(from op)
op

Processor Summary

Processors consist of

- Muxes combining
  - Registers
  - Arithmetic circuits (like the adder we showed previously)
- With inputs selected based on parts of an instruction
  - Which is bits read from memory
  - At an address from the PC
    - A register
    - Incremented each clock tick
    - Sometimes assigned a new value by a jump instruction

Other things processors do

- `push` and `pop`
  - one register points to top of stack
  - these actions both (a) load/store from top of stack and (b) change where top is
- `call` and `return`
  - `call` is both (a) `push` address of next instruction and (b) `jump`
  - `return` `pop`s address into the PC instead of into a program register
- `syscall`
  - Switches from for code to the operating system's code
  - A bit like `call`, with other complexity we'll discuss later
- Operating-system only instructions, such as
  - Receive data from other hardware (keyboard, mouse, etc)
  - Send data to other hardware (disk, network, screen, etc)
  - Change which process is running