HARVARD
School of Engineering
and Applied Sciences

Processor Architecture

CS61, Lecture 8

Prof. Stephen Chong

September 27, 2011

Announcements

- HW 3 released
  - Please check you have a directory "hw3" in your CS 61 home directory
  - Contact course staff if you don't
- Infrastructure
  - Please try not to create more than one VM at a time
- Interested in going to grad school in CS?
  - Panel tonight: 6pm in MD 119
  - Pizza

Today

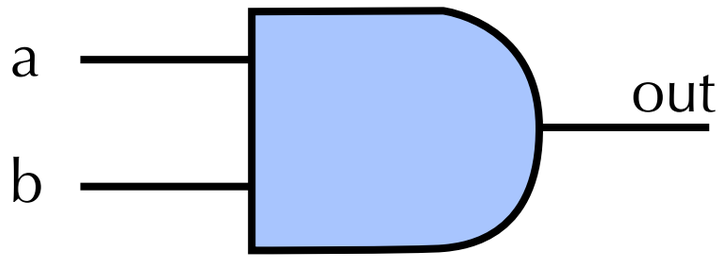
- Processor architecture
  - Logic gates
  - Adders and multiplexors
  - Registers
  - Instruction set encoding
  - A sequential processor
  - Pipelining
  - CISC vs RISC

Logic gates

- Electronic circuits used to compute functions on bits, and store bits
  - Typical technology:
    - Logical value 1 represented by high voltage (~1.0V)
    - Logical value 0 represented by low voltage (~0.0V)
- **Logic gates** are the basic computing elements of digital circuits
  - Implement Boolean functions
  - Can be implemented with transistors, electro-mechanical relays, optical components, ...

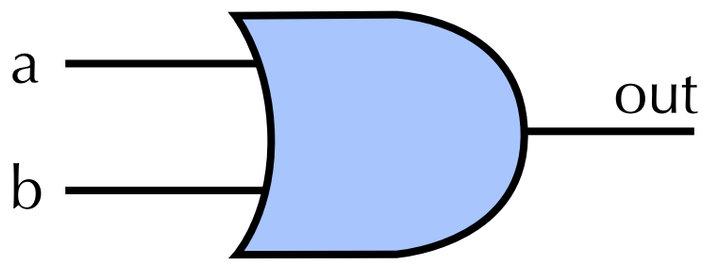
Logic gates

AND gate



a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

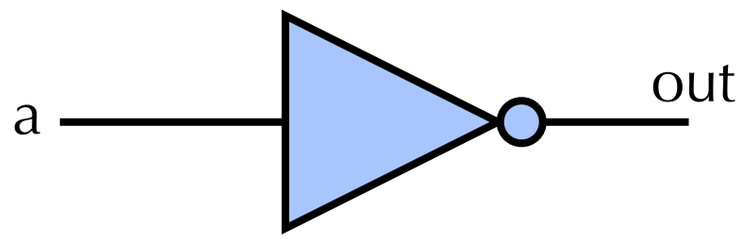
OR gate



a	b	out
0	0	0
0	1	1
1	0	1
1	1	1

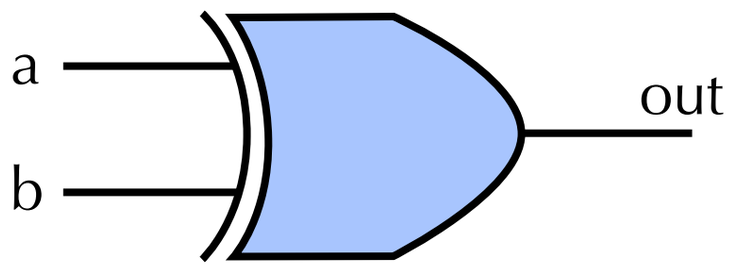
Logic gates

NOT gate



a	out
0	1
1	0

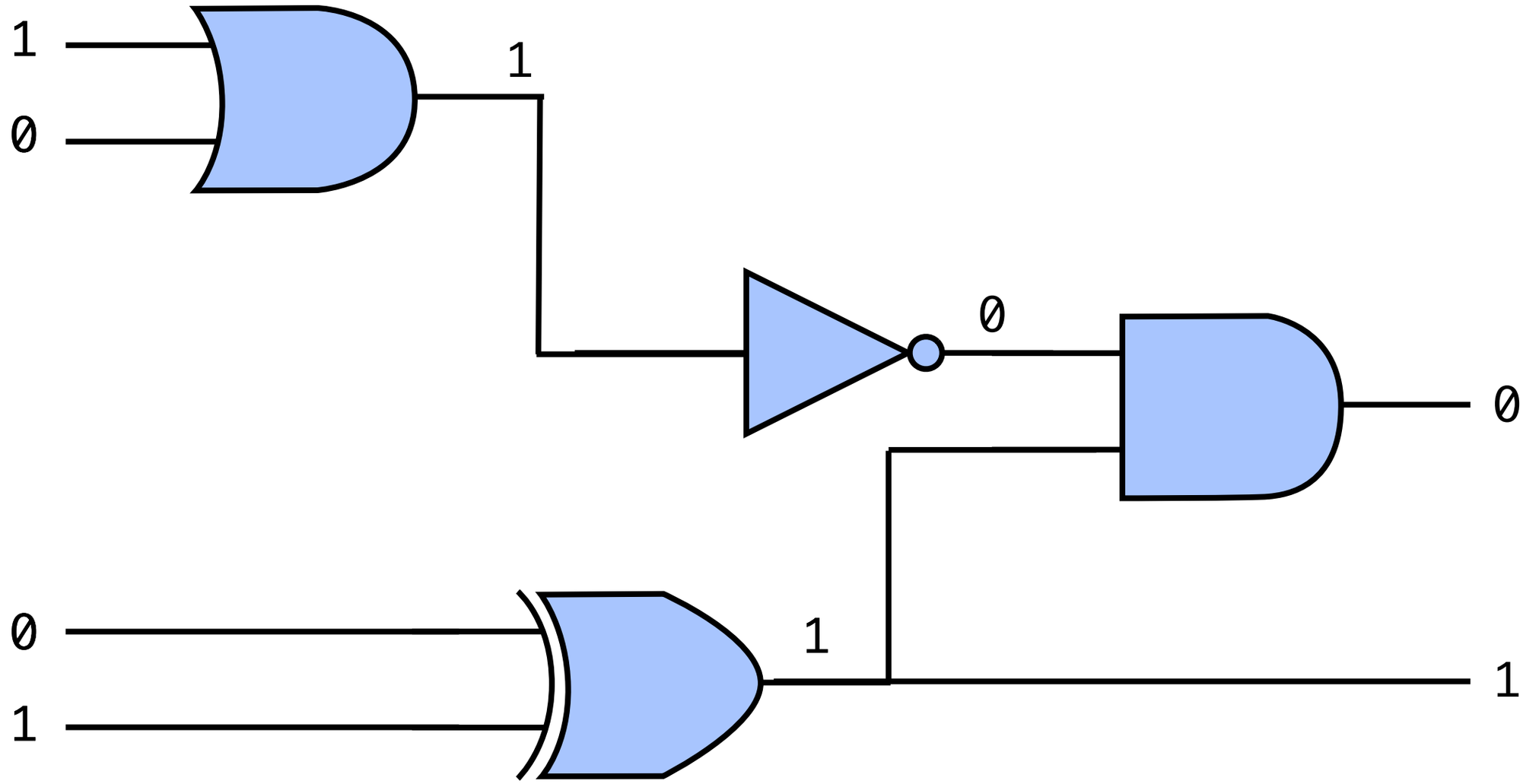
XOR gate



a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

- Other logic gates are possible
  - e.g., NAND, NOR

Logic gate example



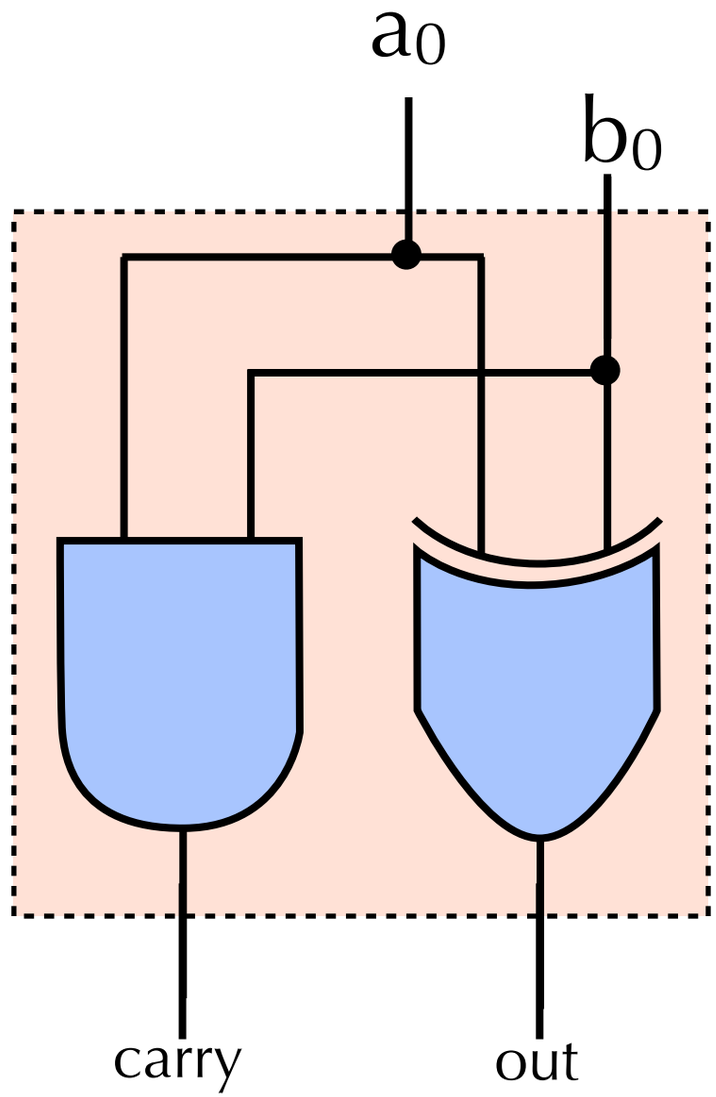
- Logic gates are always active
  - Output always being computed from input
- But takes time for signal to propagate
  - If input changes, will take some time before output correctly reflects input

A one-bit adder

- Suppose we are building a circuit to add two bits
  - $a_0 + b_0$
- How many outputs do we need?

```
  0      0      1      1
+ 0    + 1    + 0    + 1
  0      1      1     10
```

- Two: one "output" bit, and one "carry" bit

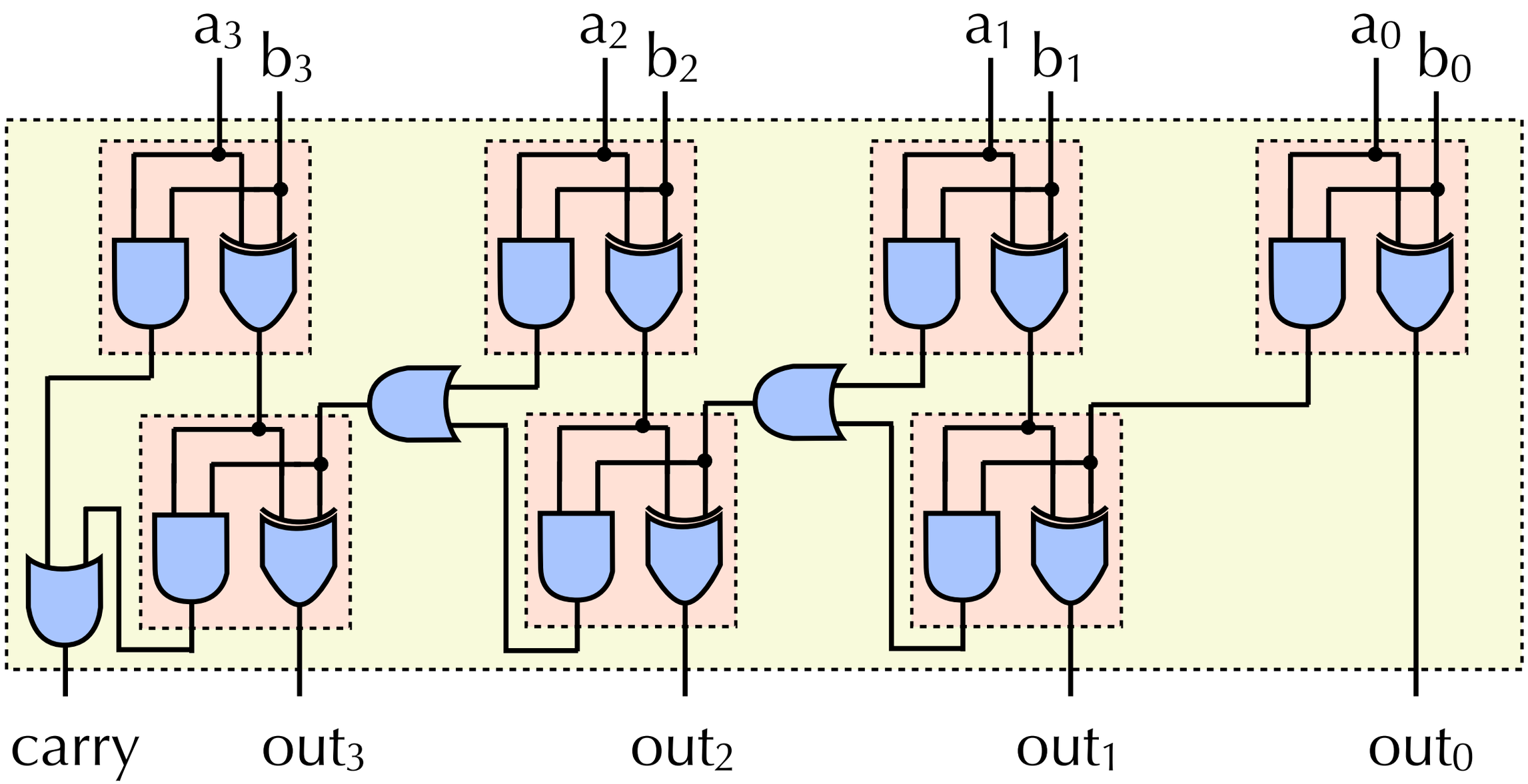


A 4-bit adder

- Suppose we are adding two 4-bit numbers
  - $a_3a_2a_1a_0 + b_3b_2b_1b_0$
- We can construct a 4-bit adder using the 1-bit adder we just developed...

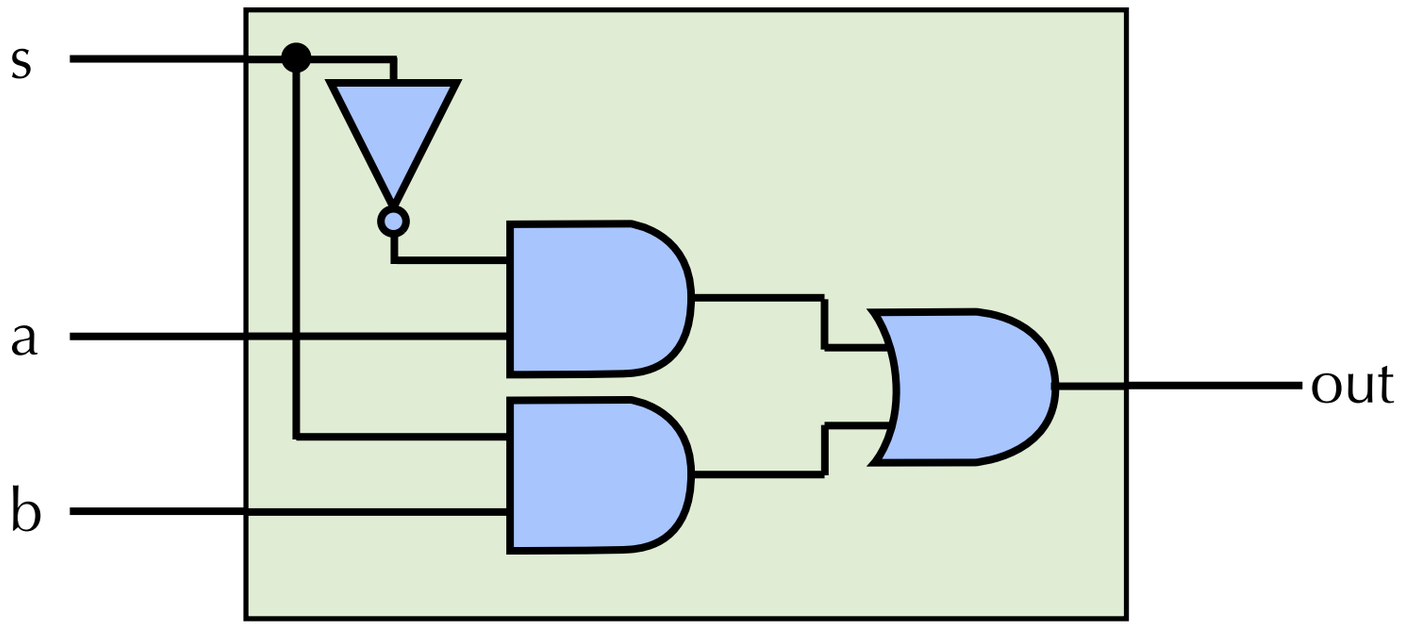
A 4-bit adder

- Suppose we are adding two 4-bit numbers



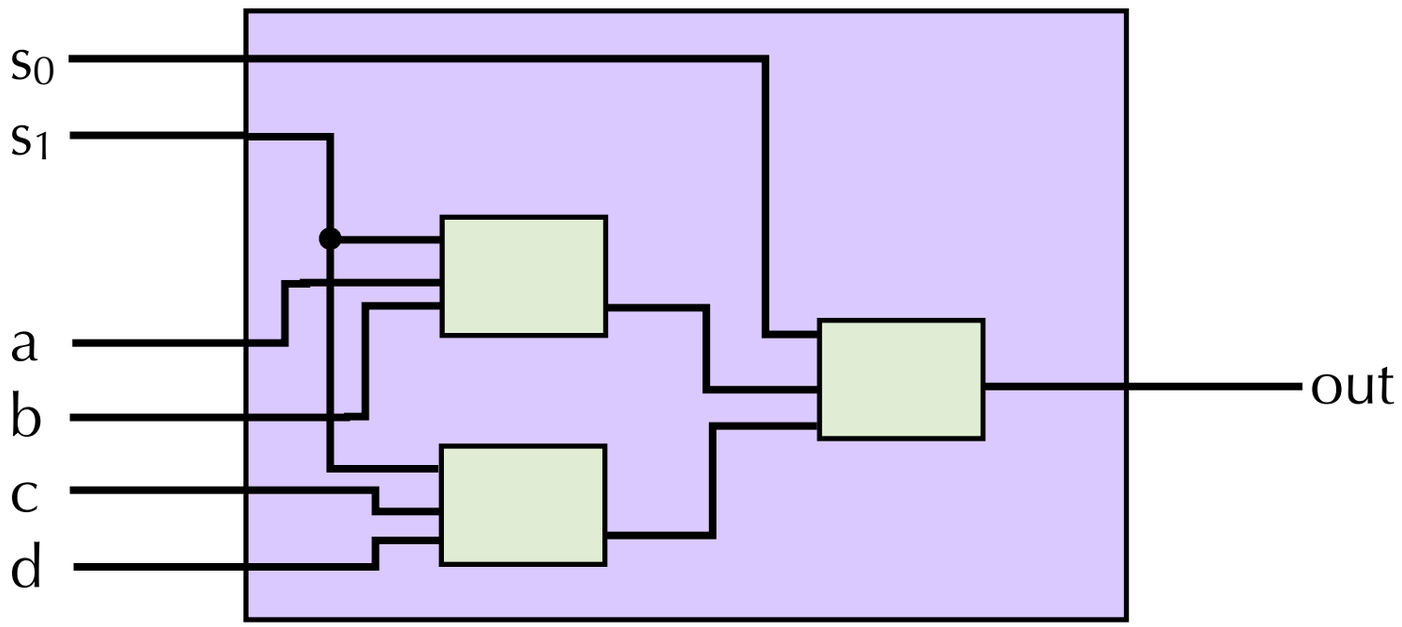
Multiplexors

- A **multiplexor** (or MUX) selects one of several data inputs based on a control input
- E.g., a single-bit multiplexor
  - If s is 0 then out will equal a
  - If s is 1 then out will equal b



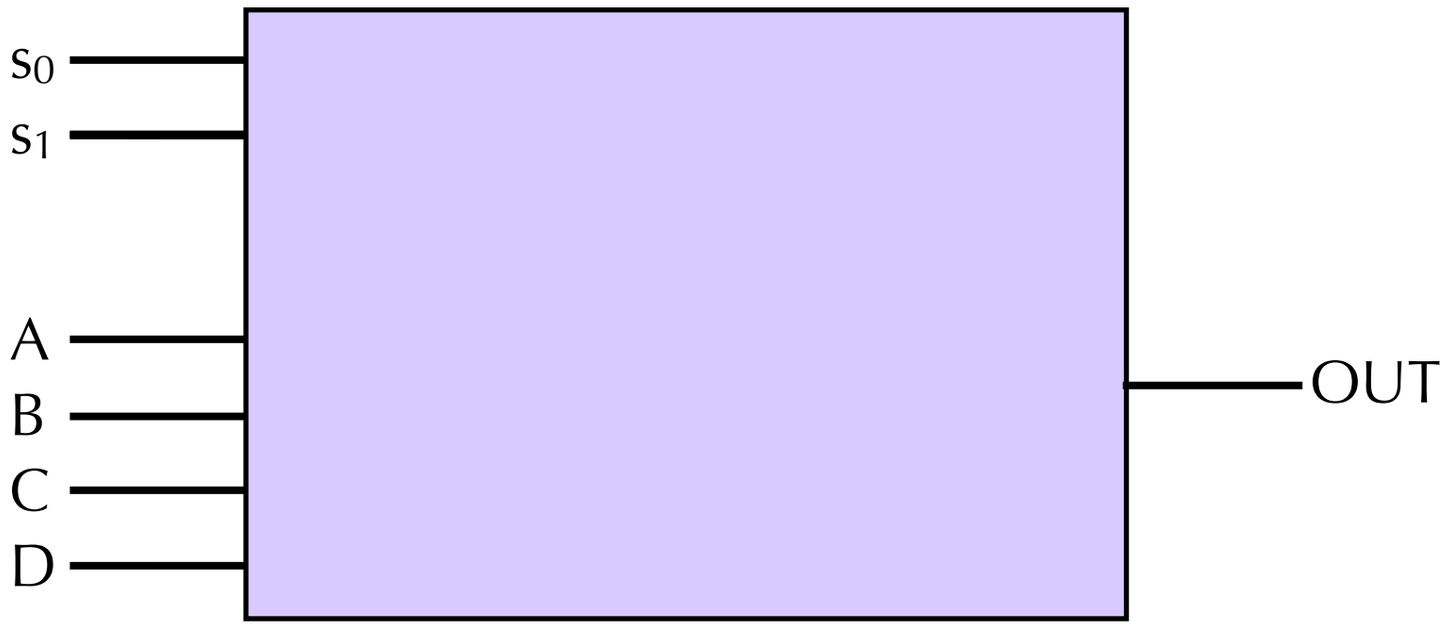
4-way multiplexor

- Can have more than just two inputs…
  - If s_0 is 0 and s_1 is 0 then out will equal a
  - If s_0 is 0 and s_1 is 1 then out will equal b
  - If s_0 is 1 and s_1 is 0 then out will equal c
  - If s_0 is 1 and s_1 is 1 then out will equal d



Word-level multiplexors

- Can also choose an entire word, not just one bit
  - A, B, C, D, and OUT represent 32-bit words
  - s_0 and s_1 select one of the words



Memory

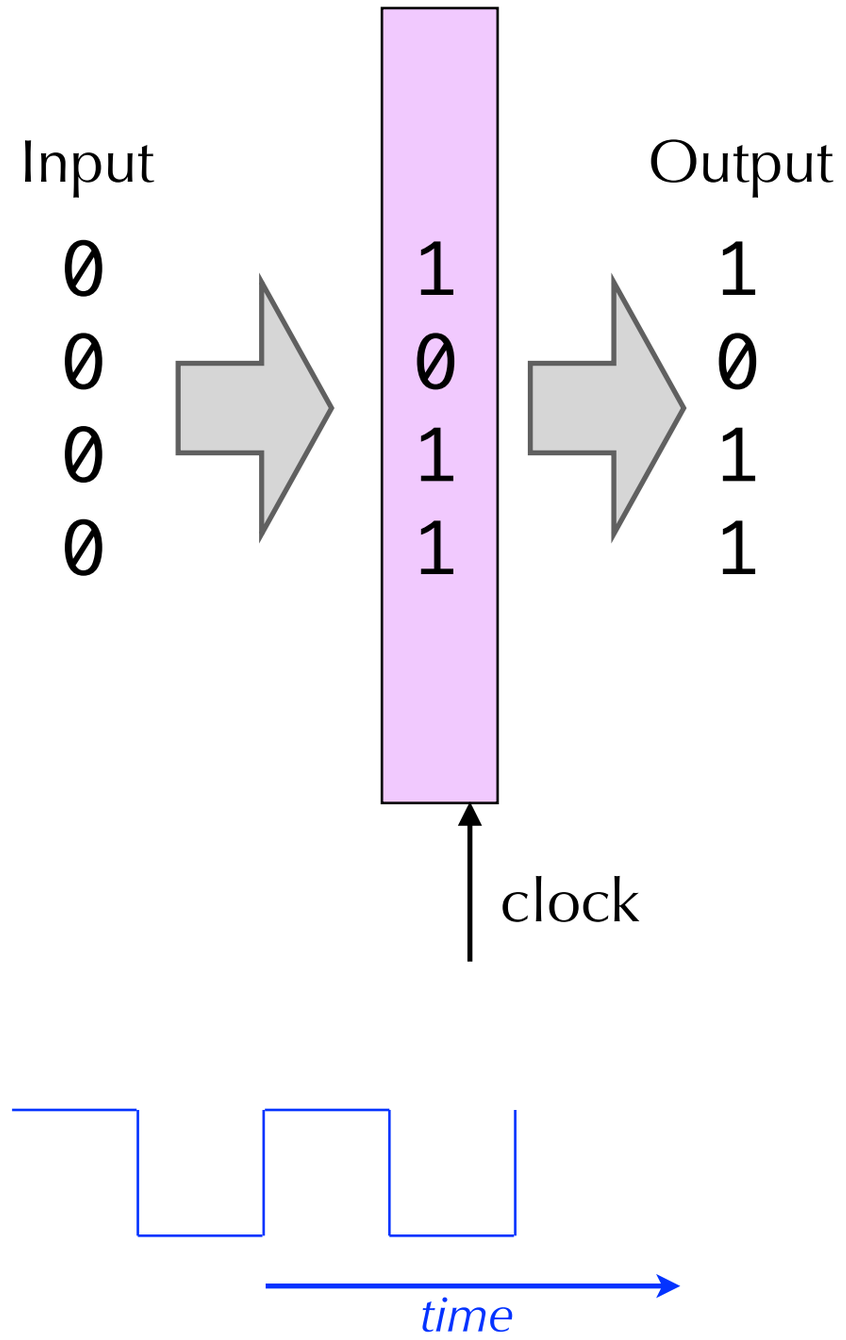
- Combinational circuits do not store information
  - They generate output based on their input
  - Do not have **state**
- Let's consider two classes of memory devices
  - **Clocked registers**
  - **Random-access memories**

Clocked registers

- **Clocked registers** (or **registers**) store individual bits or words
- A clock signal controls loading value into a register
- Note: slightly different meaning of word *register*
  - In hardware: *register* is storage directly connected to rest of circuit by its input and output wires
  - In machine-level programming: *register* refers to small collection of addressable words in the CPU
  - Program registers can be implemented with hardware registers

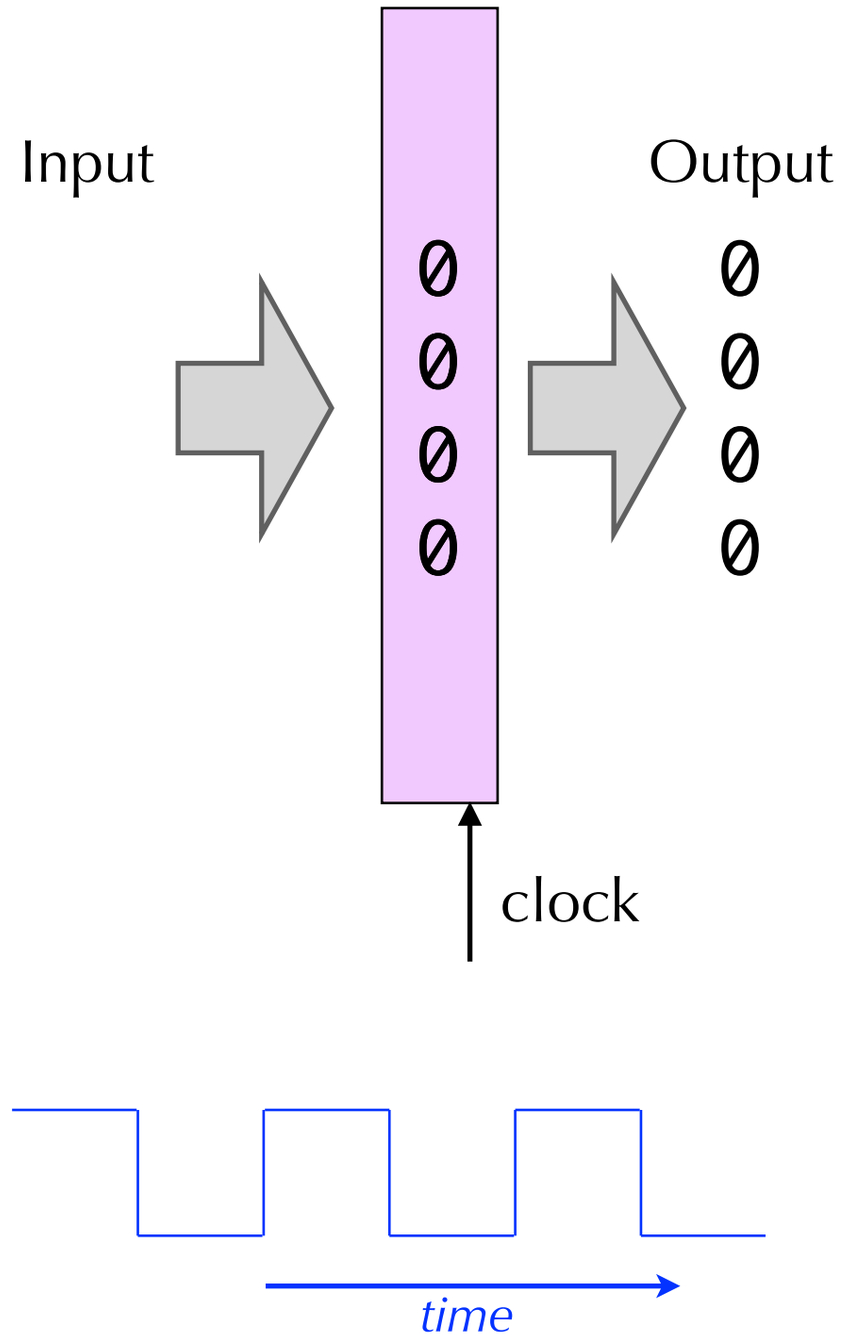
Register

- Register output is the current register state
- When clock rises, values at register input is captured, and becomes new state



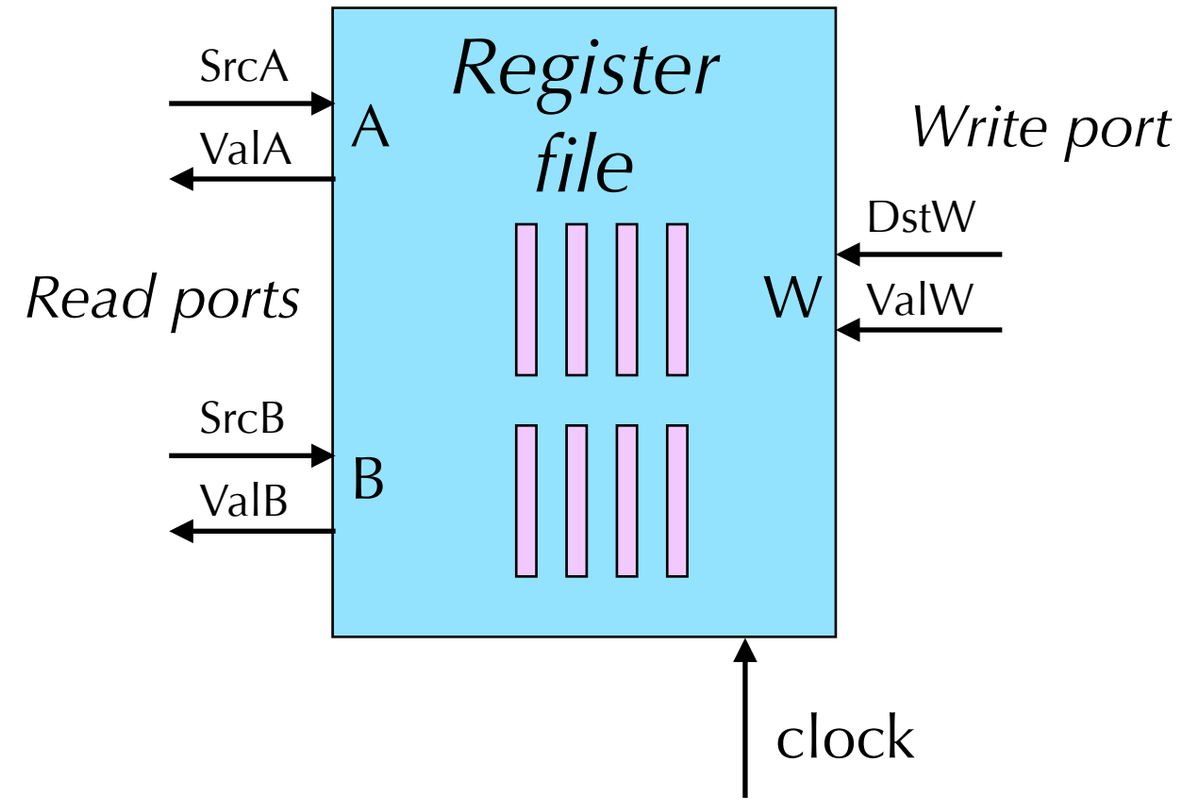
Register

- Register output is the current register state
- When clock rises, values at register input is captured, and becomes new state



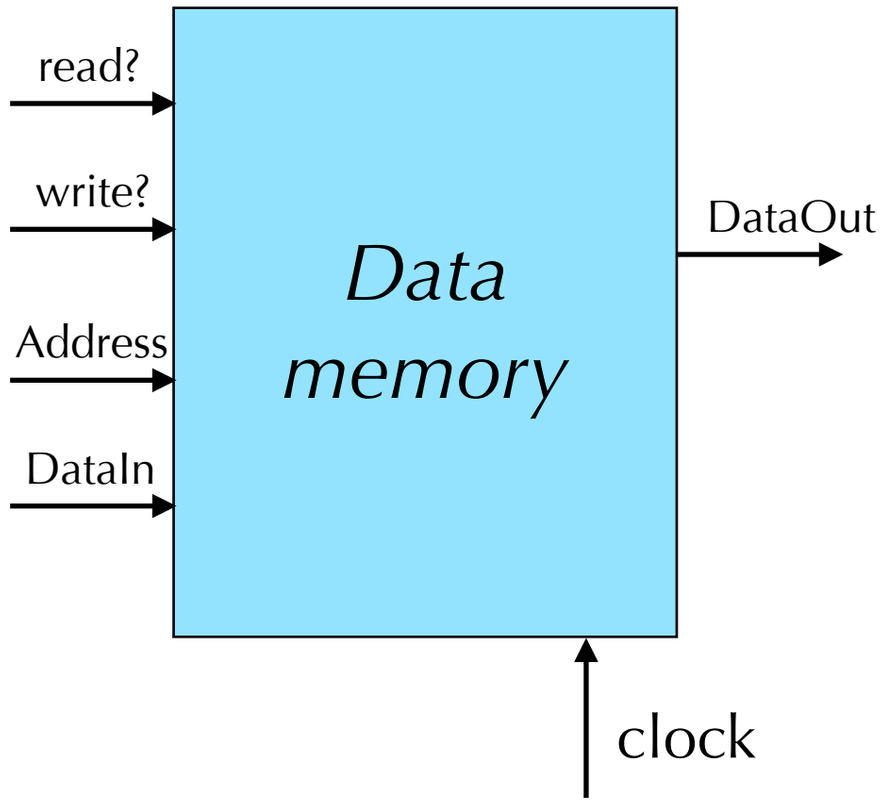
Register file

- A **register file** has multiple registers
  - Can perform simultaneous reads and writes
- E.g.,
  - Two read ports A and B
    - SrcA selects one of the registers, the contents of which are output on ValA
    - E.g., SrcA set to `011` selects program register `%ebx`
  - One write port W
    - DstW selects one of the registers, the contents of which are overwritten with ValW on the clock rise



Data memory

- For the moment, let's assume a simple model for main memory
  - Specify an Address
  - If read?=1 then contents of address will be on DataOut
  - If write?=1 then contents of address will be set to DataIn on clock rise
- Memory is more complicated than this
  - We will see more about memory later in the course



How do they fit together?

- Suppose we encode instruction

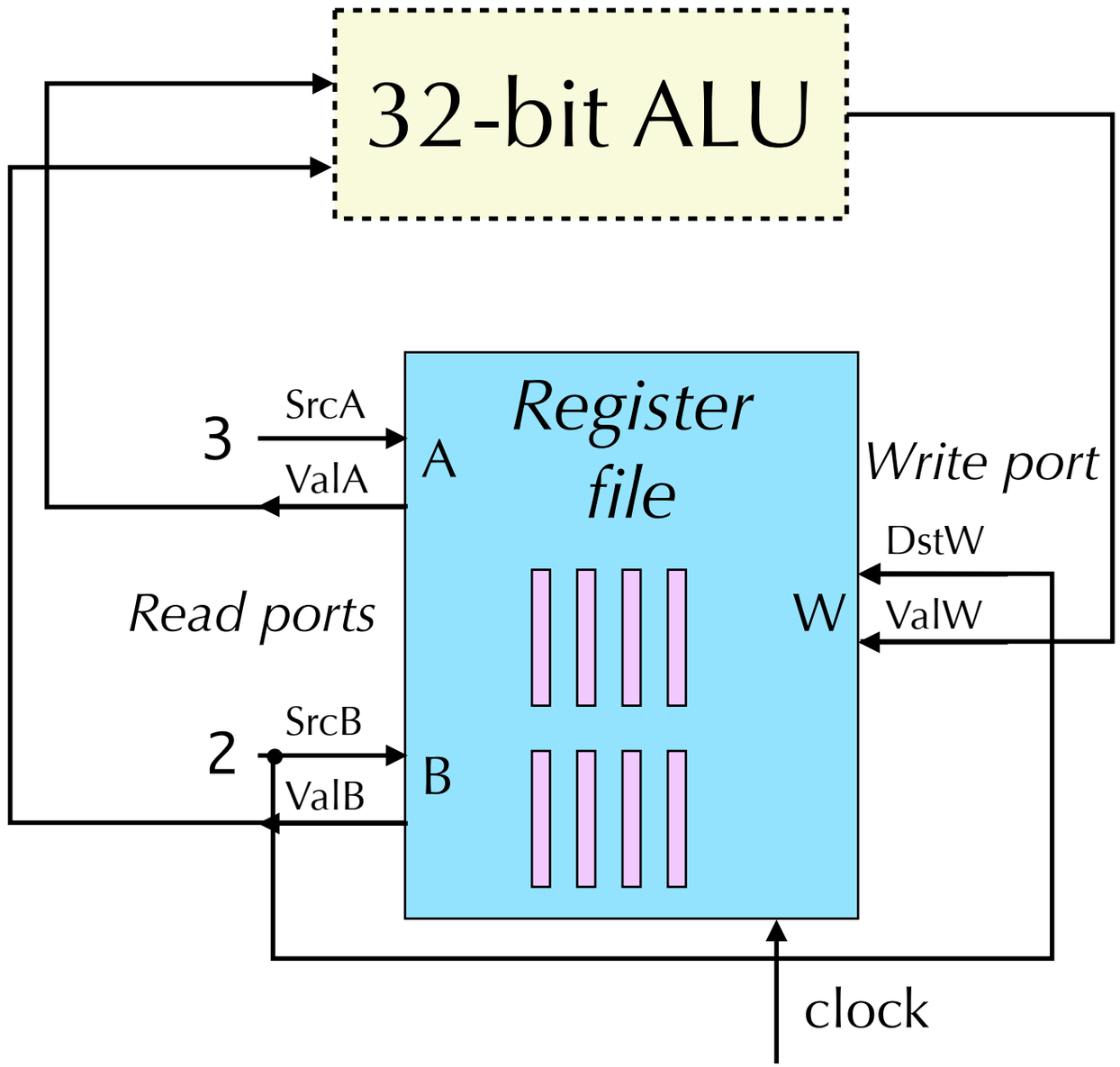
  `add %ebx, %edx`

as bytes `0x6032`

Addition instruction

%ebx

%edx



Today

- Processor architecture
  - Logic gates
  - Adders and multiplexors
  - Registers
  - Instruction set encoding
  - A sequential processor
  - Pipelining
  - CISC vs RISC

Instruction Set Architectures

- Many different instructions, even in a simple ISA
  - Move instructions
    - Different variants: to and from register, from memory to register, from register to memory
  - Arithmetic/logical instructions
    - `add`, `sub`, `and`, `xor`, ...
  - Control flow instructions
    - `jmp`, `jz`, `jle`, `jl`, ...
    - `call`, `ret`
  - ...
- What instructions are in the ISA influence how we can implement the ISA
- We'll consider a simple instruction set: Y86
  - Described in text book; a simplification of x86

Instruction encoding

- Machine instructions are encoded as bytes

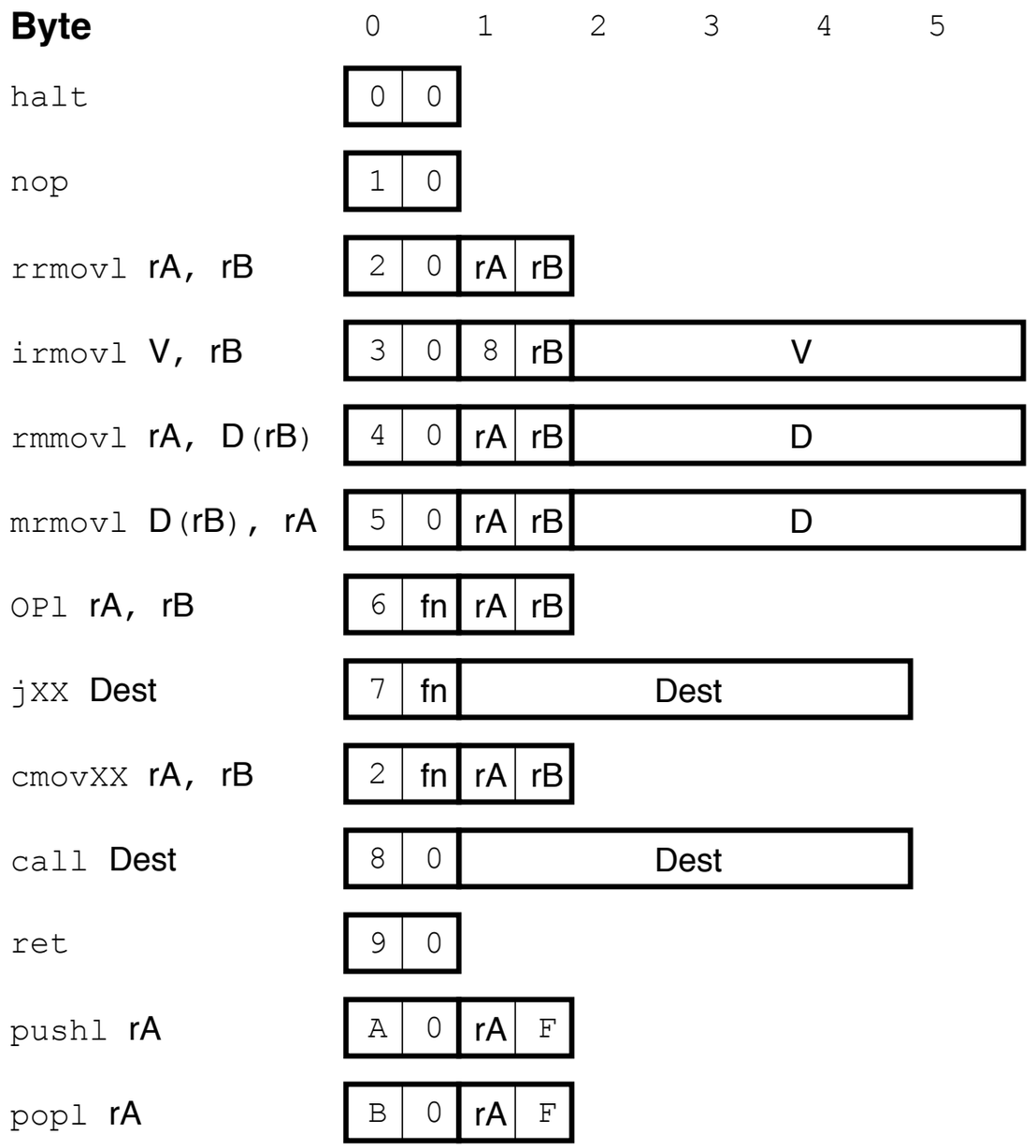
```
% objdump -d scshell
...
 8048404:     55                      push   %ebp
 8048405:     89 e5                   mov    %esp,%ebp
 8048407:     83 ec 14                sub    $0x14,%esp
 804840a:     c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
 8048411:     eb 2a                   jmp    804843d
 8048413:     8b 15 4c 97 04 08       mov    0x804974c,%edx
 8048419:     8b 45 fc                mov    -0x4(%ebp),%eax
 804841c:     8d 04 02                lea    (%edx,%eax,1),%eax
 804841f:     0f b6 10                movzbl (%eax),%edx
 8048422:     8b 45 08                mov    0x8(%ebp),%eax
 8048425:     0f b6 00                movzbl (%eax),%eax
 8048428:     38 c2                   cmp    %al,%dl
 804842a:     74 09                   je     8048435
...
```

Instruction encoding

- Machine instructions are encoded as bytes
- How instructions are encoded affects how we implement an ISA
- We use a *variable length* encoding
  - Different instructions may take a different number of bytes
  - x86 is a variable length encoding
  - More compact representation that fixed length, but more complex to figure out the address of the next instruction

Instruction encoding

- Initial byte specifies instruction type
  - High-order nibble indicates code, and low-order nibble indicates function
    - E.g., `60` is `addl`, `61` is `subl`
- Registers are identified using 4 bits



Sequential processor

- Let's first try a processor that executes one instruction each clock cycle
  - Requires a long clock cycle (why?) ⇒ inefficient
  - We will improve performance soon…
- Break each instruction up into 6 stages
  - Uniform structure simplifies implementation
  - Not every instruction will have each stage

Execution stages

Fetch

Read bytes of instruction from memory

Decode

Reads up to two operands from the register file

Execute

Perform arithmetic/logic operation

(includes computing effective address of memory reference, incrementing or decrementing stack pointer, testing condition codes for branch conditions, …)

Memory

Read or write data to/from memory

Write back

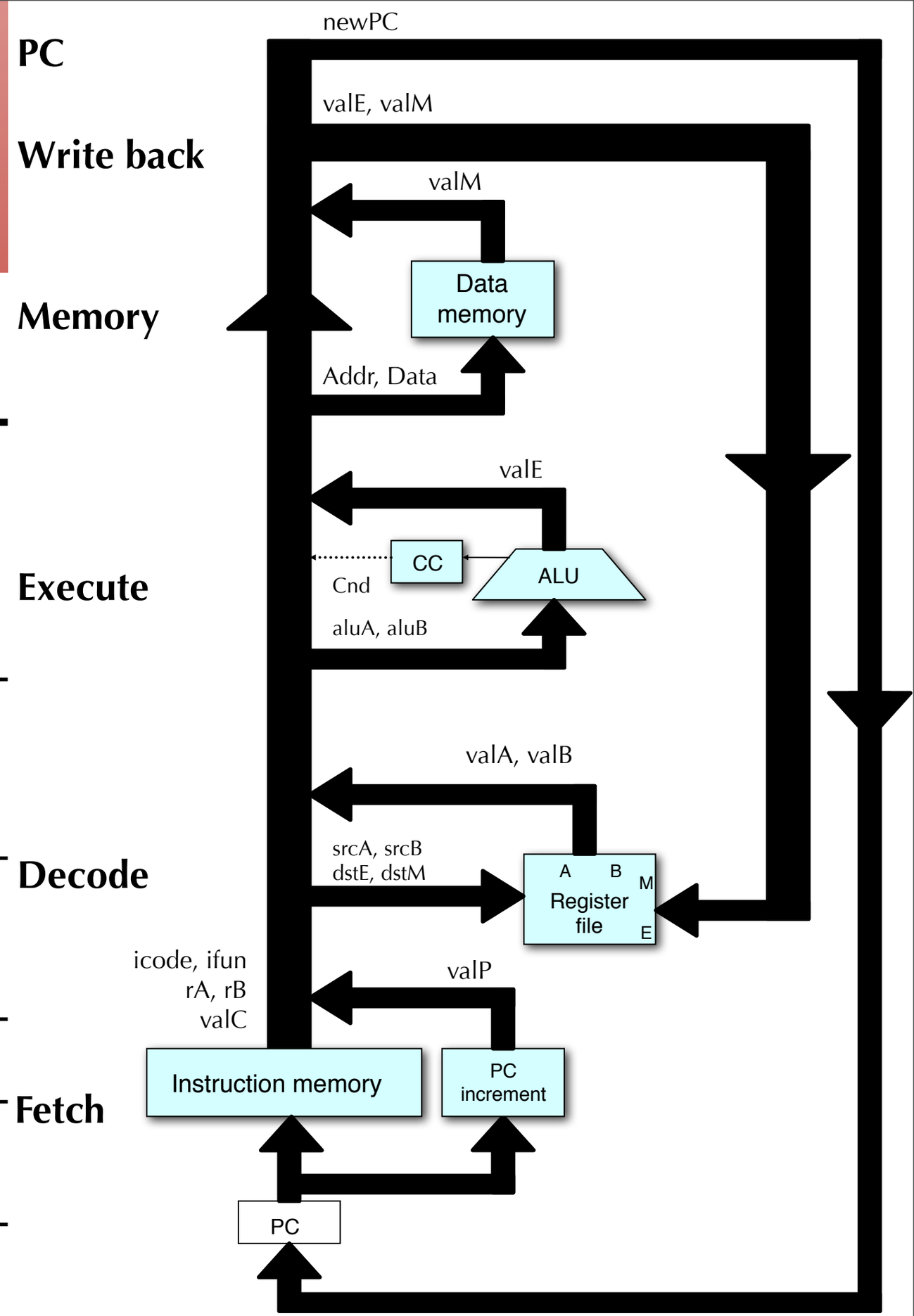
Writes up to two results to the register file

PC update

Set PC to address of next instruction

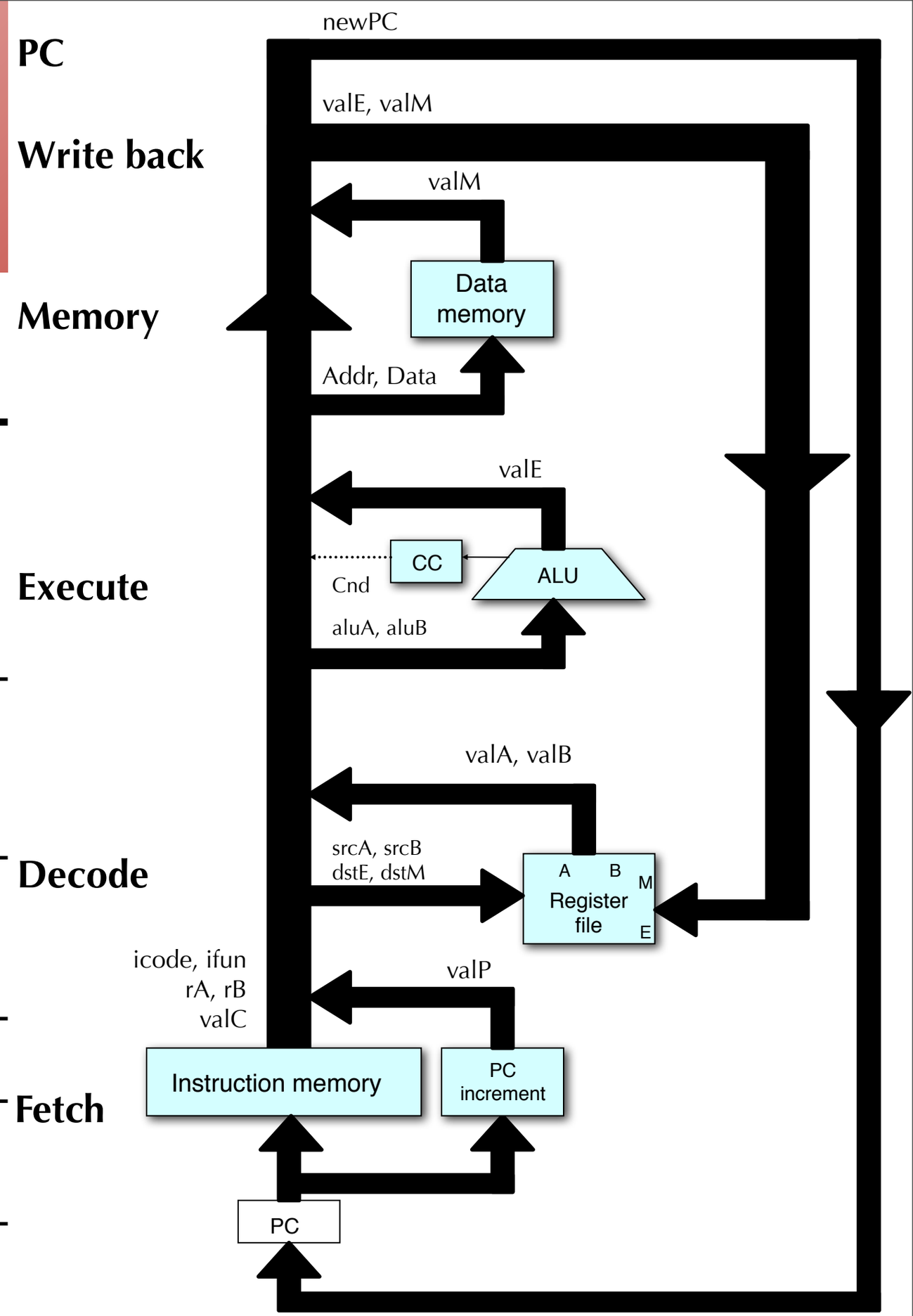
Execution stages

Stage	OPl rA, rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2
Decode	valA ← R[rA] valB ← R[rB]
Execute	valE ← valB OP valA Set CC
Memory	–
Write back	R[rB] ← valE
PC update	PC ← valP



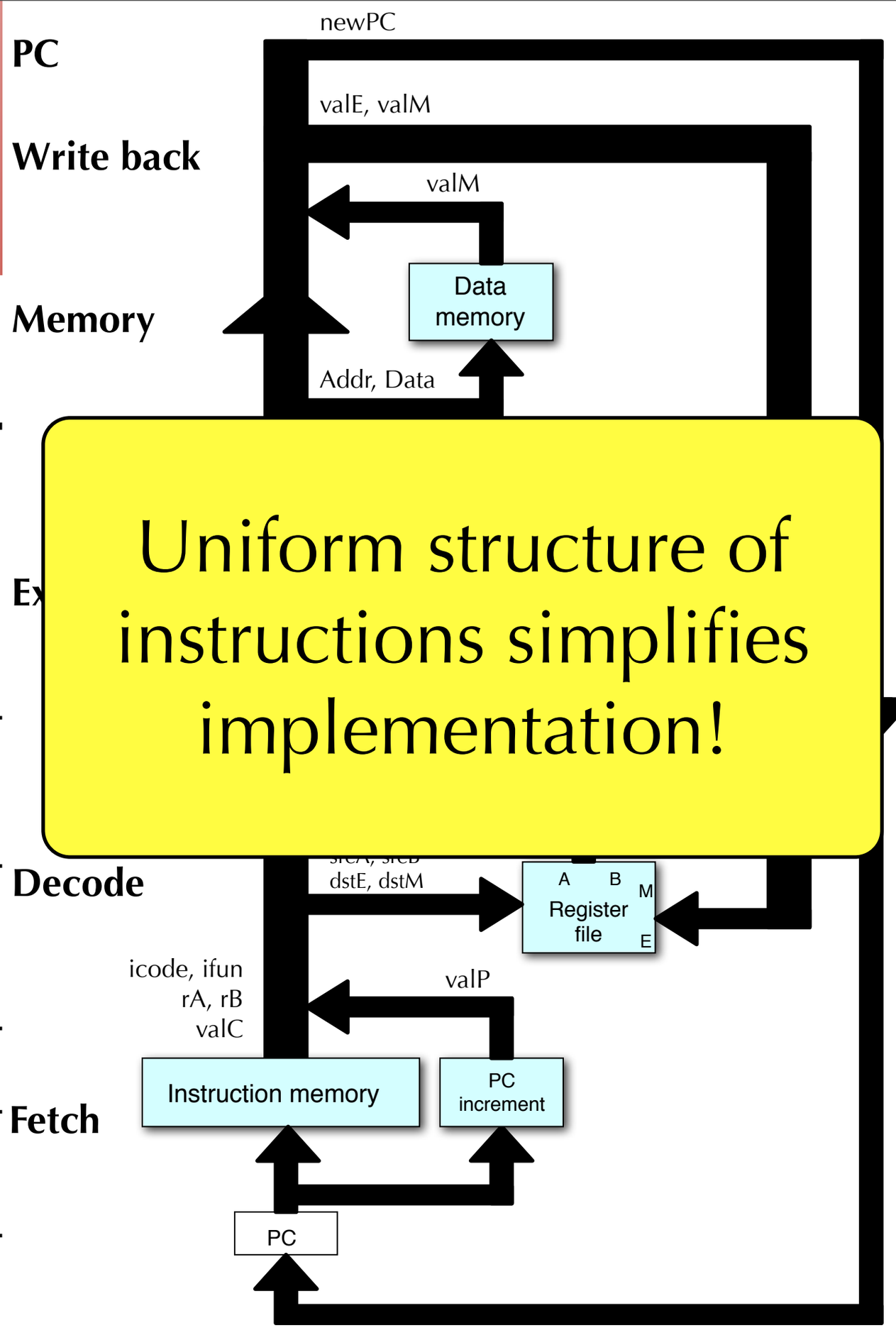
Execution stages

Stage	rrmovl rA, rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2
Decode	valA ← R[rA]
Execute	valE ← valA
Memory	–
Write back	R[rB] ← valE
PC update	PC ← valP



Execution stages

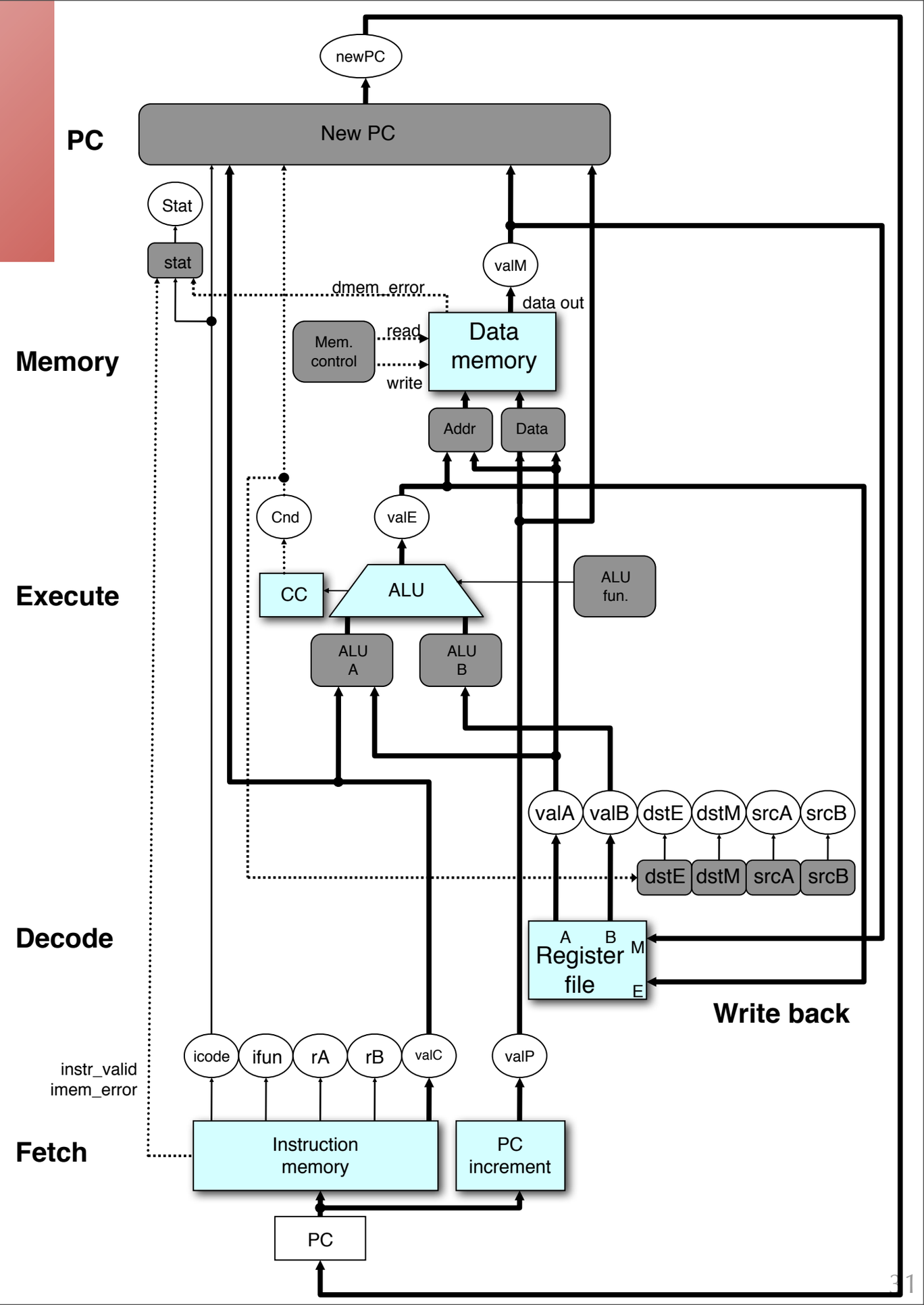
Stage	mrmovl C(rB), rA
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC + 6
Decode	valB ← R[rB]
Execute	valE ← valB + valC
Memory	valM ← M_4[valE]
Write back	R[rA] ← valM
PC update	PC ← valP



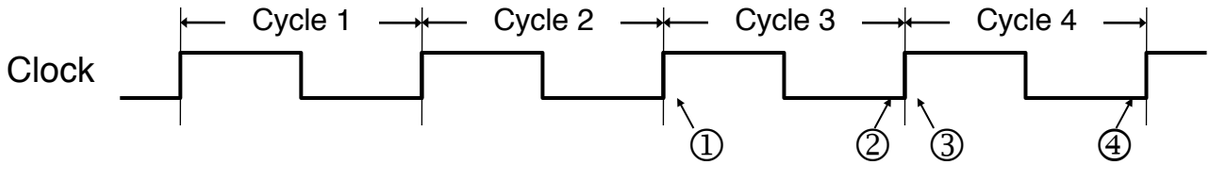
Uniform structure of instructions simplifies implementation!

Hardware structure

- Processing is performed by **hardware units** associated with the different stages
- Clock cycle needs to be long enough that signal can propagate throughout all units

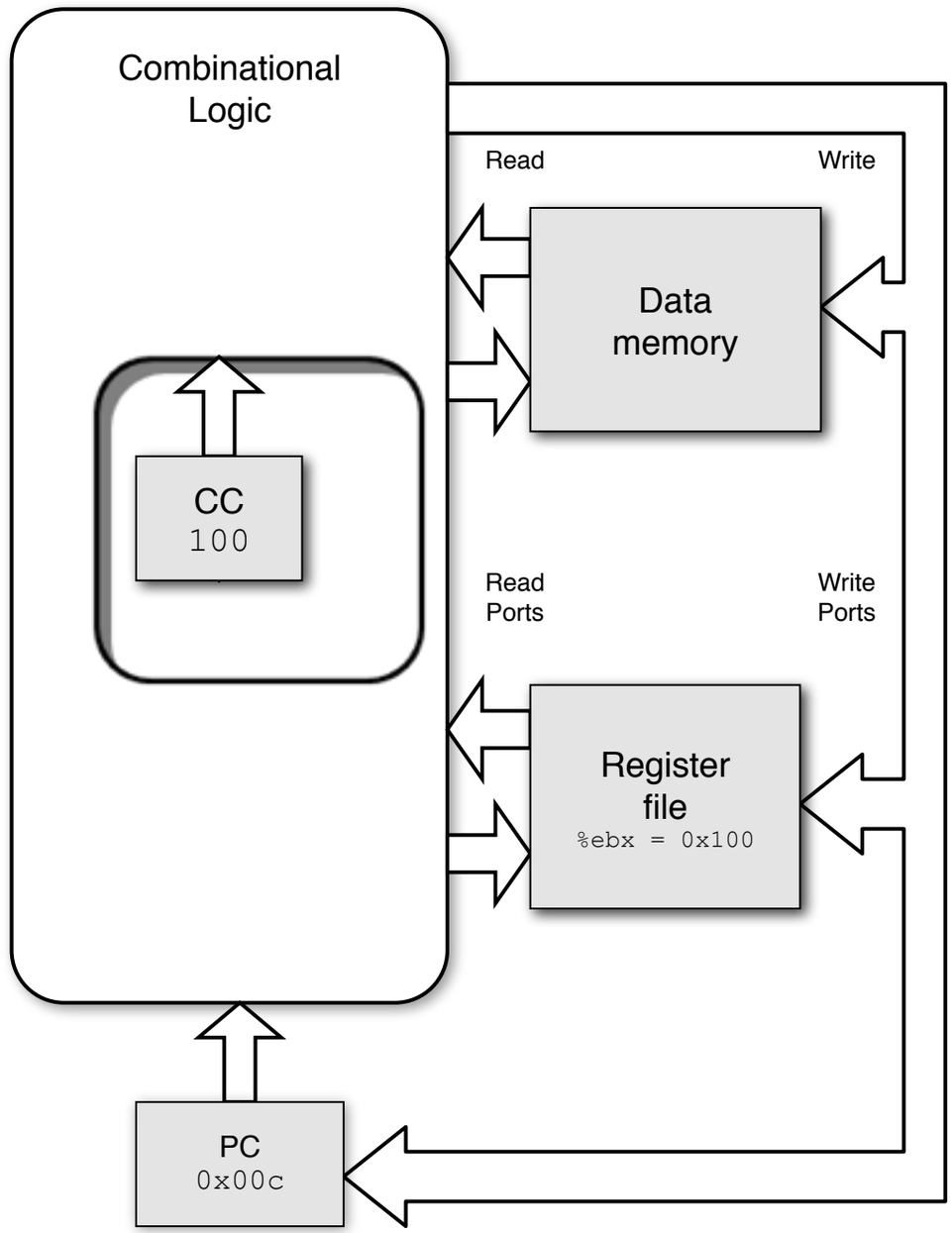


Tracing cycles

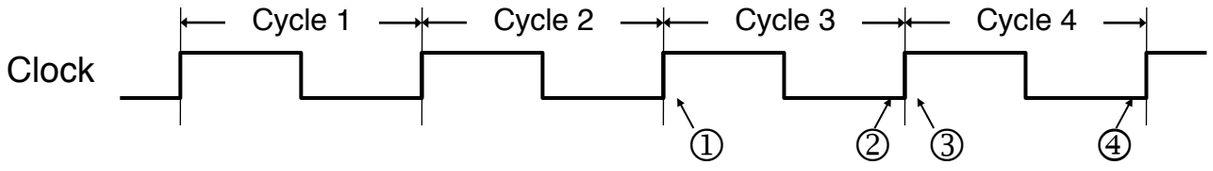


Cycle 1:	0x000:	irmovl $0x100,%ebx	# %ebx <-- 0x100
Cycle 2:	0x006:	irmovl $0x200,%edx	# %edx <-- 0x200
Cycle 3:	0x00c:	addl %edx,%ebx	# %ebx <-- 0x300 CC <-- 000
Cycle 4:	0x00e:	je dest	# Not taken
Cycle 5:	0x013:	rmmovl %ebx,0(%edx)	# M[0x200] <-- 0x300

① Beginning of cycle 3

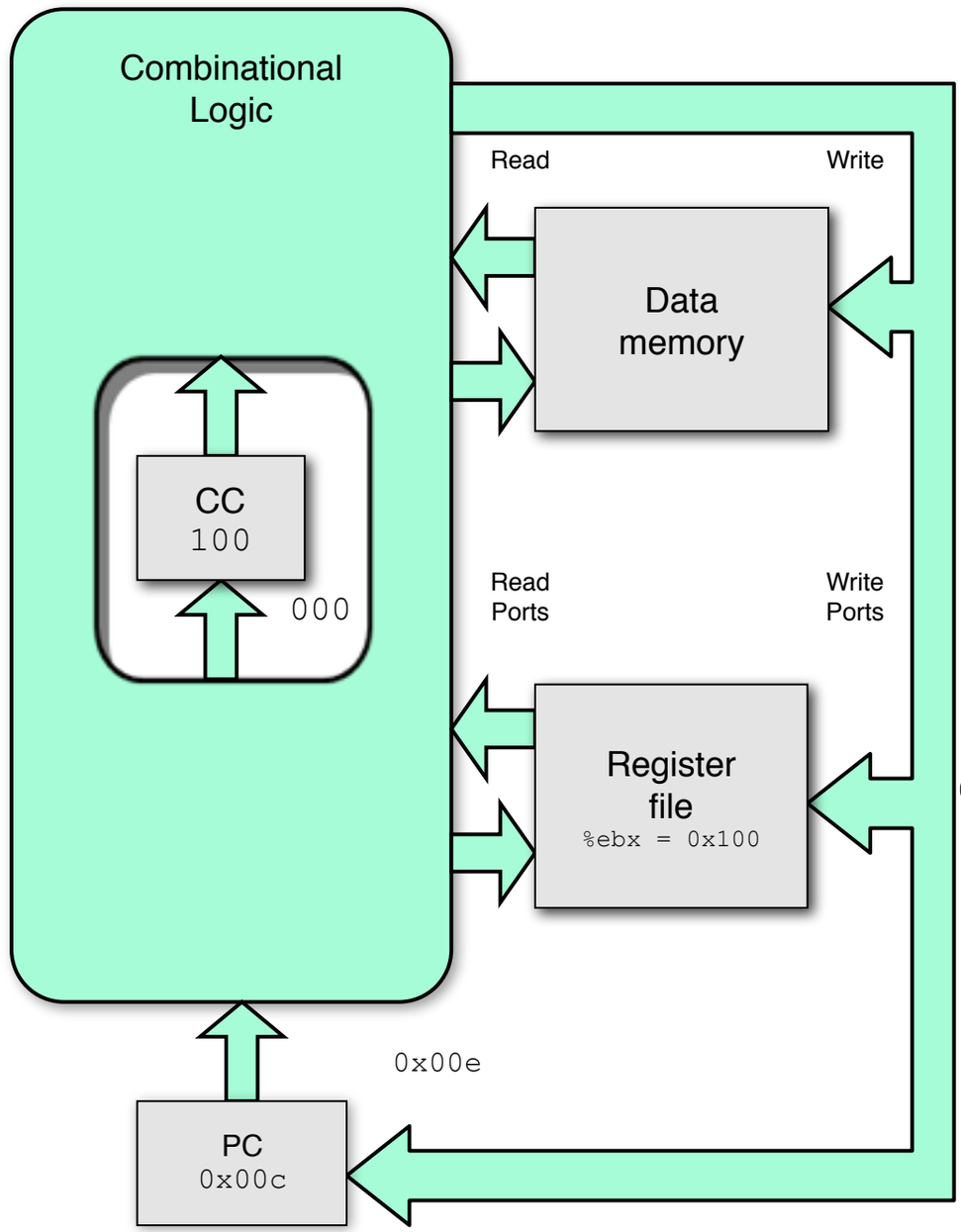


Tracing cycles

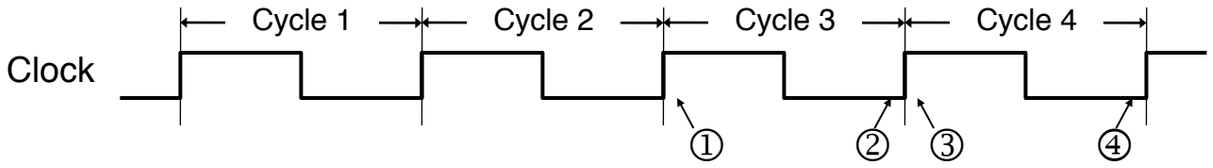


Cycle 1:	0x000:	irmovl $0x100,%ebx	# %ebx <-- 0x100
Cycle 2:	0x006:	irmovl $0x200,%edx	# %edx <-- 0x200
Cycle 3:	0x00c:	addl %edx,%ebx	# %ebx <-- 0x300 CC <-- 000
Cycle 4:	0x00e:	je dest	# Not taken
Cycle 5:	0x013:	rmmovl %ebx,0(%edx)	# M[0x200] <-- 0x300

② End of cycle 3

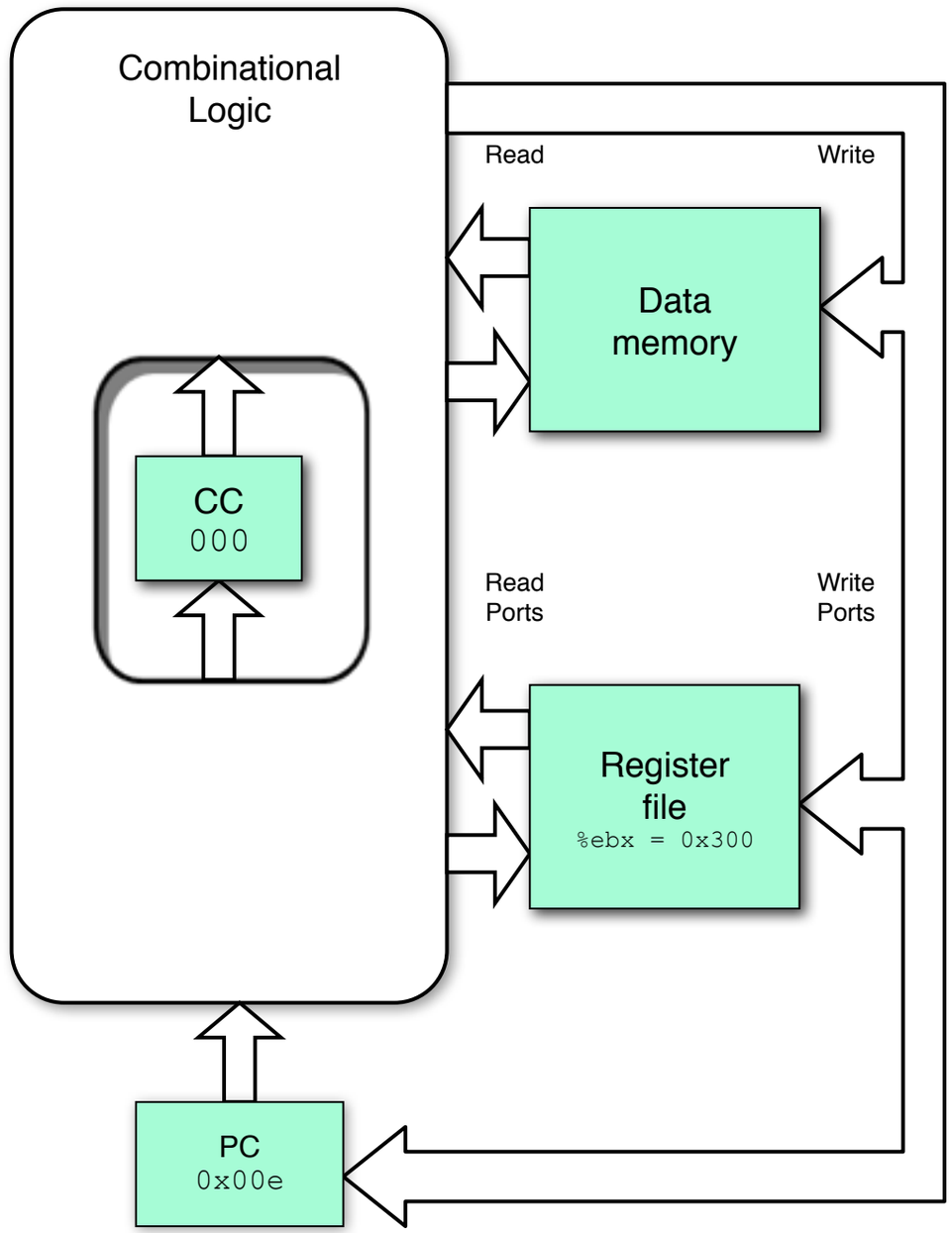


Tracing cycles

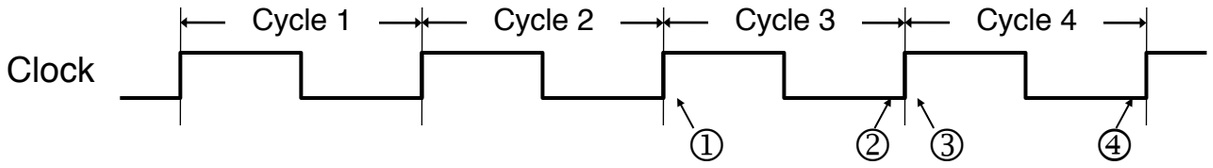


Cycle 1:	0x000:	irmovl $0x100,%ebx	# %ebx <-- 0x100
Cycle 2:	0x006:	irmovl $0x200,%edx	# %edx <-- 0x200
Cycle 3:	0x00c:	addl %edx,%ebx	# %ebx <-- 0x300 CC <-- 000
Cycle 4:	0x00e:	je dest	# Not taken
Cycle 5:	0x013:	rmmovl %ebx,0(%edx)	# M[0x200] <-- 0x300

③ Beginning of cycle 4

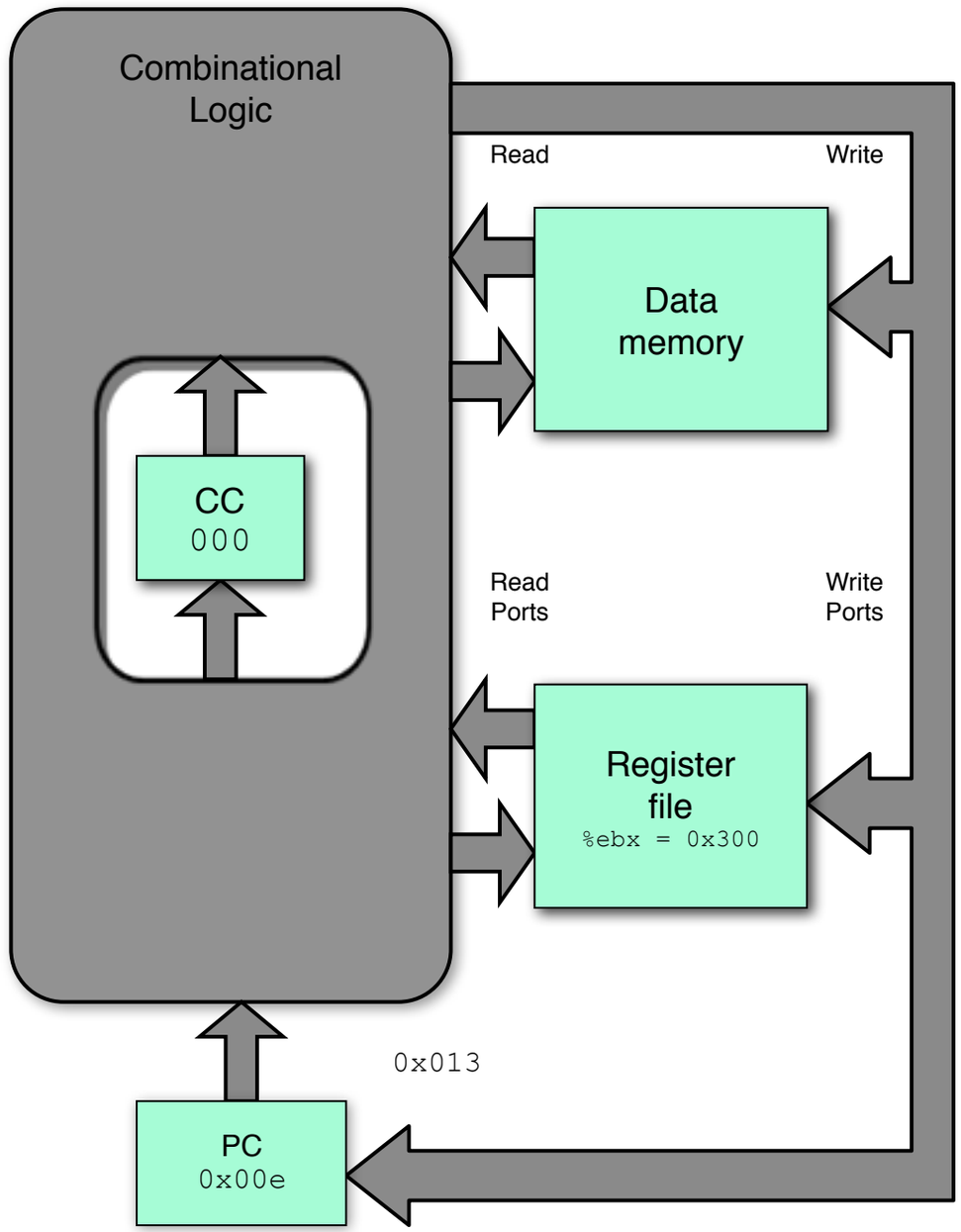


Tracing cycles



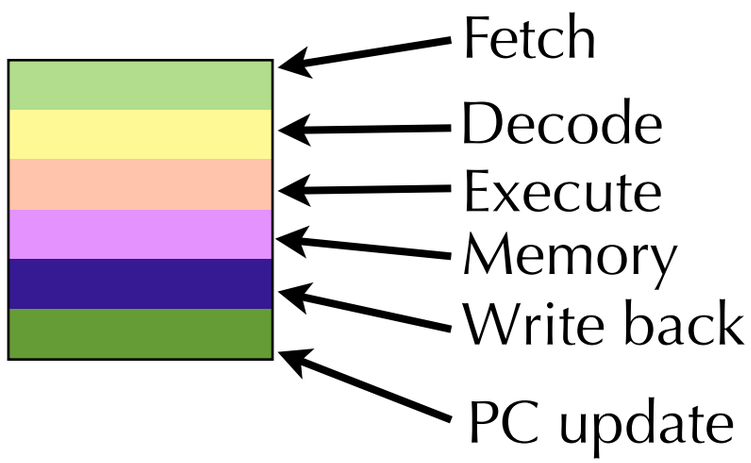
Cycle 1:	0x000:	irmovl $0x100,%ebx	# %ebx <-- 0x100
Cycle 2:	0x006:	irmovl $0x200,%edx	# %edx <-- 0x200
Cycle 3:	0x00c:	addl %edx,%ebx	# %ebx <-- 0x300 CC <-- 000
Cycle 4:	0x00e:	je dest	# Not taken
Cycle 5:	0x013:	rmmovl %ebx,0(%edx)	# M[0x200] <-- 0x300

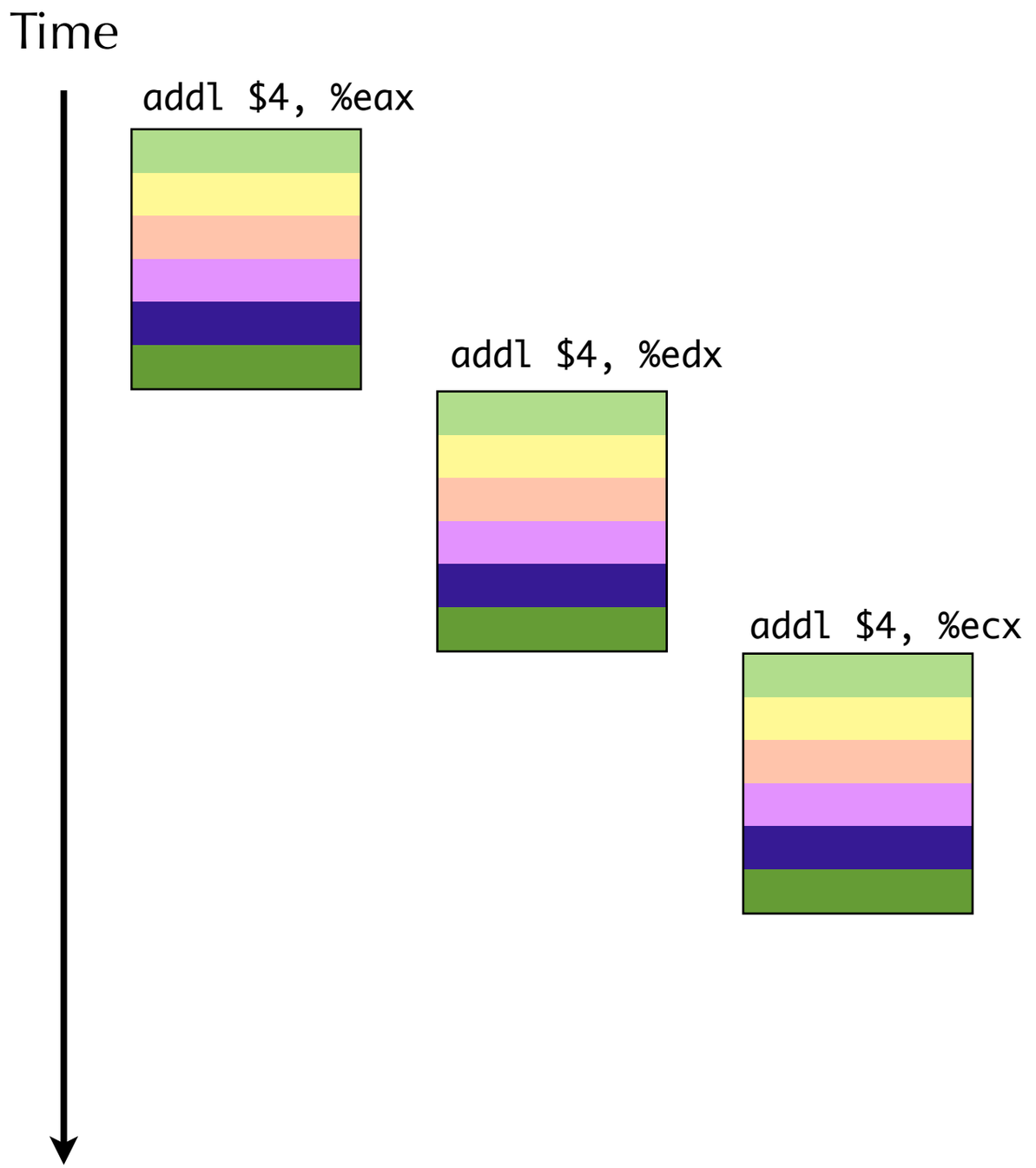
④ End of cycle 4



Inefficient!

- Because cycle has to be long enough for signal to propagate, hardware units are often idle!
  - Only one instruction is executed at a time

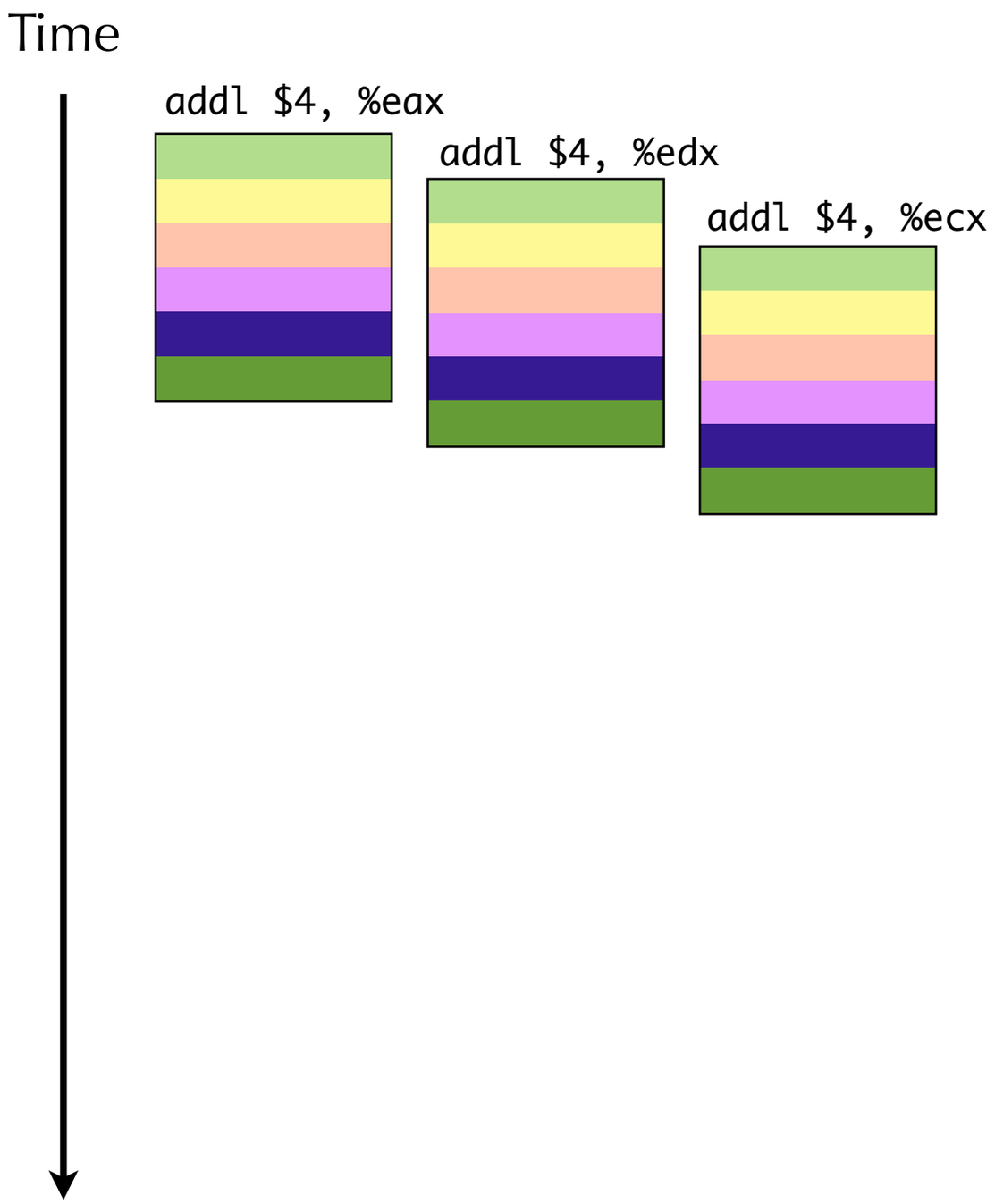




Pipelining

- But we can start on next instruction while previous instruction is still processing
- This is know as **pipelining**:
  - Task to be performed is divided into discrete stages
  - Multiple tasks can be worked on at same time, each in a different stage
  - E.g., Car wash
    - Suppose 3 stages: Suds, Rinse, and Polish
- Pipelining increases **throughput** of system
  - Number of tasks per unit time is higher
- Pipelining may increase **latency**
  - Any one task may take longer than if no pipeline used

Pipelining



- Clock can be much faster
  - Needs to be long enough for signal to propagate through a single stage
- In order to achieve pipelining, need to add hardware registers between the hardware units
  - Overhead of these registers limits how deep a pipeline can be

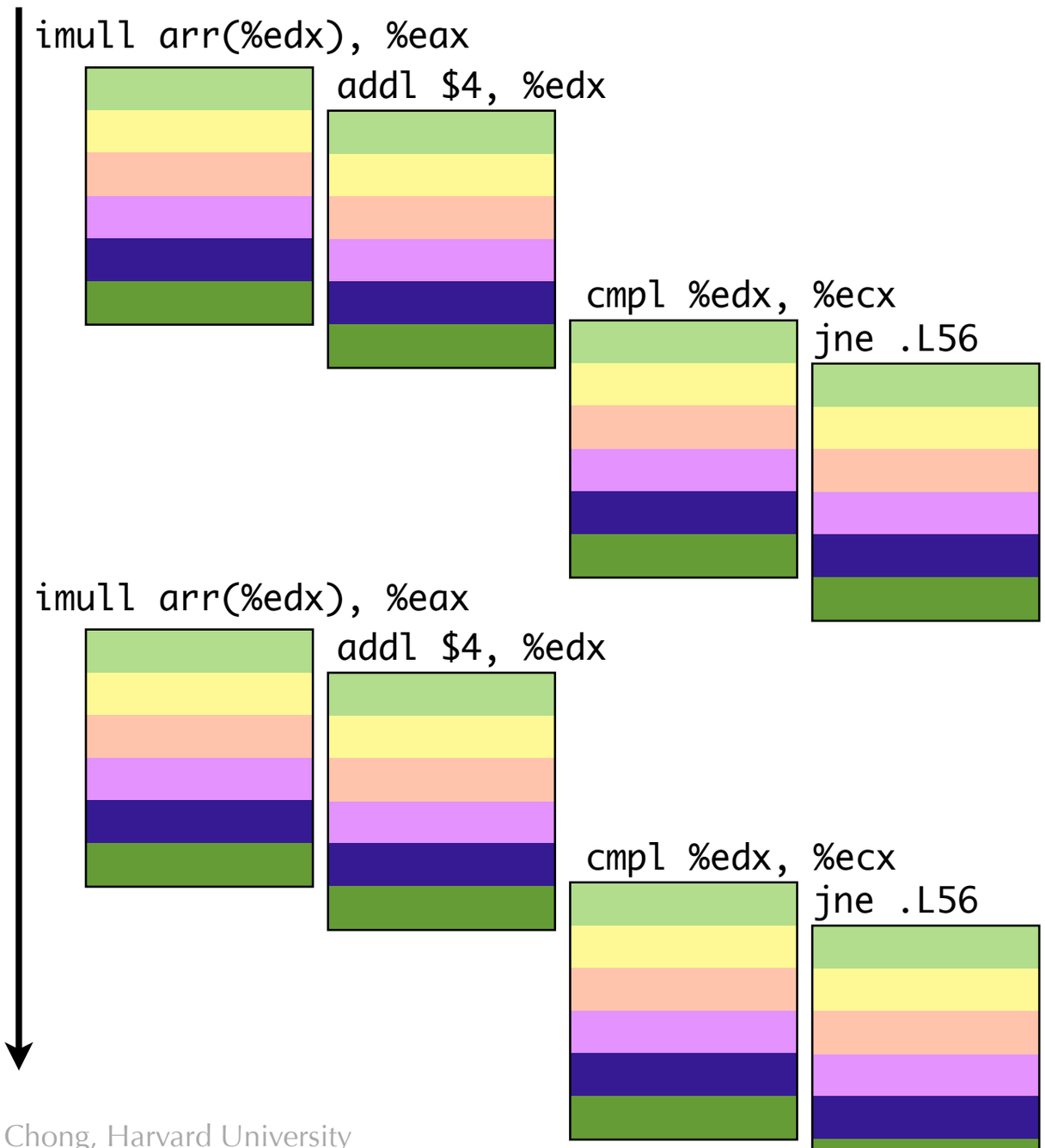
Example

```
int * arr;
int prod_array(int n) {
  int i, result=1;
  for (i = 0; i < n; i++) {
    result *= arr[i];
  }
  return result;
}
```

```
    # Main body of loop
    .L56:
        imull   arr(%edx), %eax
        addl    $4, %edx
        cmpl    %edx, %ecx
        jne     .L56
```

Example

Time




```
    # Main body of loop
    .L56:
        imull   arr(%edx), %eax
        addl    $4, %edx
        cmpl    %edx, %ecx
        jne     .L56
```

- `cmpl` needs to read `%edx`, which isn't written until the write-back stage of `addl $4, %edx`!
  - One solution: Insert "bubble" in the pipeline, waiting until results are available
  - Also known as **stalling**
- What PC should be used for next iteration?
  - Result not available until Update PC stage of `jne`
- Other **hazards** can also complicate pipeline operation

More efficient pipelines

- Instead of inserting bubbles into pipeline, can we do better?
- **Forwarding results**
  - Instead of waiting for write-back stage of previous instruction, previous instruction passes result directly to next instruction
- **Branch prediction**
  - Instead of waiting to know next PC, predict what it will be and start executing
  - May be wrong: need to be able to throw away speculative execution
  - Modern branch prediction techniques have very good prediction rates: 90%+
- **Out-of-order execution**
  - Re-ordering instructions can reduce stalling
  - Processor may execute instructions in different order than specified
- …

Today

- Processor architecture
  - Logic gates
  - Adders and multiplexors
  - Registers
  - Instruction set encoding
  - A sequential processor
  - Pipelining
  - CISC vs RISC

CISC vs RISC

- CISC ("sisk"): Complex Instruction Set Computer
- RISC ("risk"): Reduced Instruction Set Computer
- Different philosophies regarding the design (and implementation) of ISAs

CISC vs RISC

- CISC ("sisk"): Complex Instruction Set Computer
  - Historically first
  - Large instruction sets (x86 has several hundred)
  - Specialized instructions for high-level tasks
    - Instructions that are closer to what applications are wanting to do
    - Can provide hardware support for application-specific instructions
      - E.g., x86 contains instructions such as LOOPZ *label*, which decrements %cx (without modifying flags) and jumps to label if %cx is non-zero
- Presents a clean interface to programmer
  - Hides implementation details such as pipelining

CISC vs RISC

- RISC ("risk"): Reduced Instruction Set Computer
  - Philosophy developed in early 1980s
  - Small, simple, instruction sets (typically <100)
    - E.g., may have only base+displacement memory addressing
    - E.g., memory access only via load and store; ALU operations need register operands
    - E.g., no condition codes, only explicit test instructions
    - Often fixed length encodings for instructions
    - Leads to simple, efficient implementation
  - Reveals implementation details to programmer (e.g., pipe-lining)
    - E.g., certain instruction sequences may be prohibited
    - E.g., jump instruction may not take effect until after following instruction
      - Compiler must be aware of these restrictions, and can use them to optimize performance
  - ARM (originally "Acorn RISC Machine") widely used in embedded devices

Modern computers

- In most settings, neither CISC nor RISC clearly better
- RISC machines
  - Exposing implementation details made it difficult to use them, and difficult to evolve the ISA
  - Added more instructions
- CISC machines
  - Take advantage of RISC-like pipelines
    - Essentially translate CISC instructions into simpler RISC-like instructions
    - E.g., `addl %eax, 8(%esp)` broken up into a load from memory, followed by an addition, followed by a store to memory

Next Lecture

- Program optimizations
  - How to greatly improve the performance of your C programs with some simple changes
- Code motion
- Strength reduction
- Common subexpression elimination
- Loop unrolling
- Tail recursion