Building an Arithmetic Machine

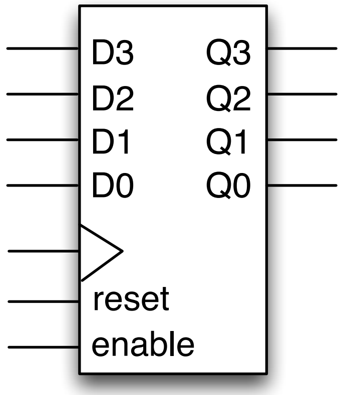
pick up HANDOUT

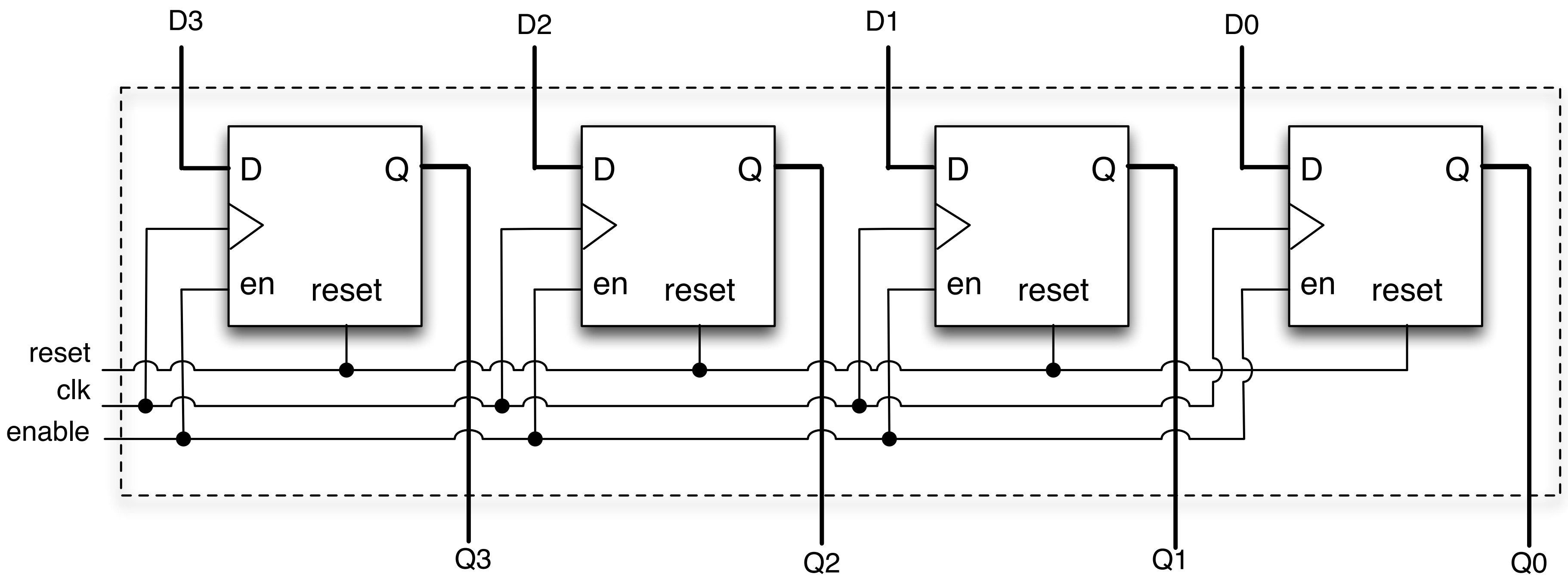
Today's lecture

- **Register Files, review/cont.**
  - Registers
- **The Arithmetic Machine**
  - Programmable hardware
  - Instruction Set Architectures (ISA) $\leftarrow$ MIPS
  - Instructions & Registers
    - Assembly Language
    - ~~Machine Language~~

How can we store more than 1 bit?

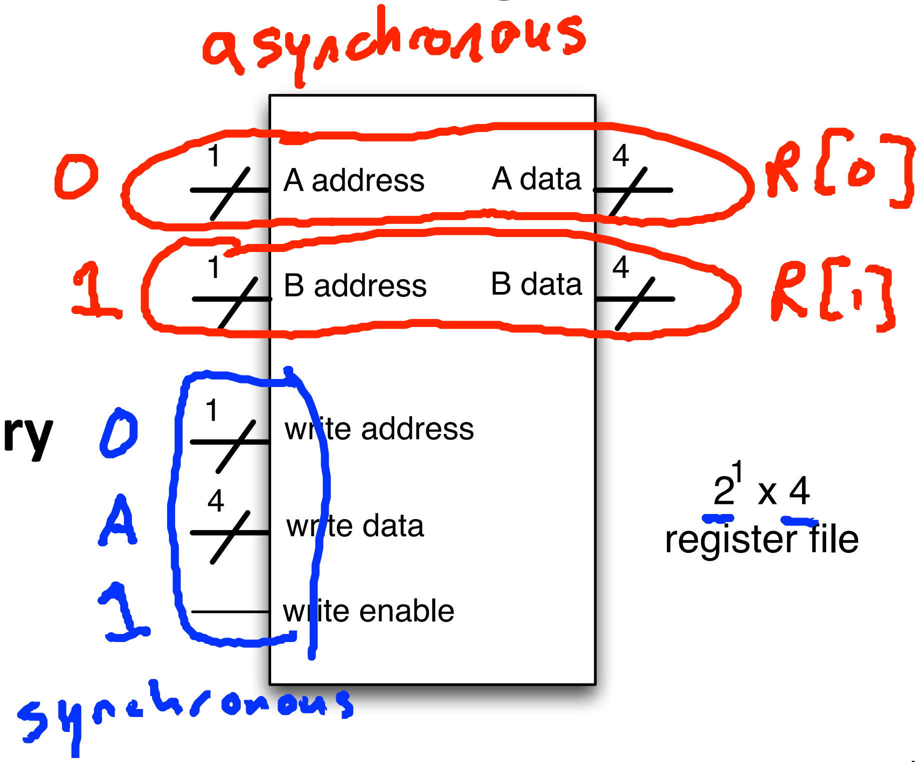
- **We build registers out of flip flops.**
  - Example 4-bit register made of four D flip flops
    - 1 data input, 1 data output per flip flop
    - All control signals use the same input





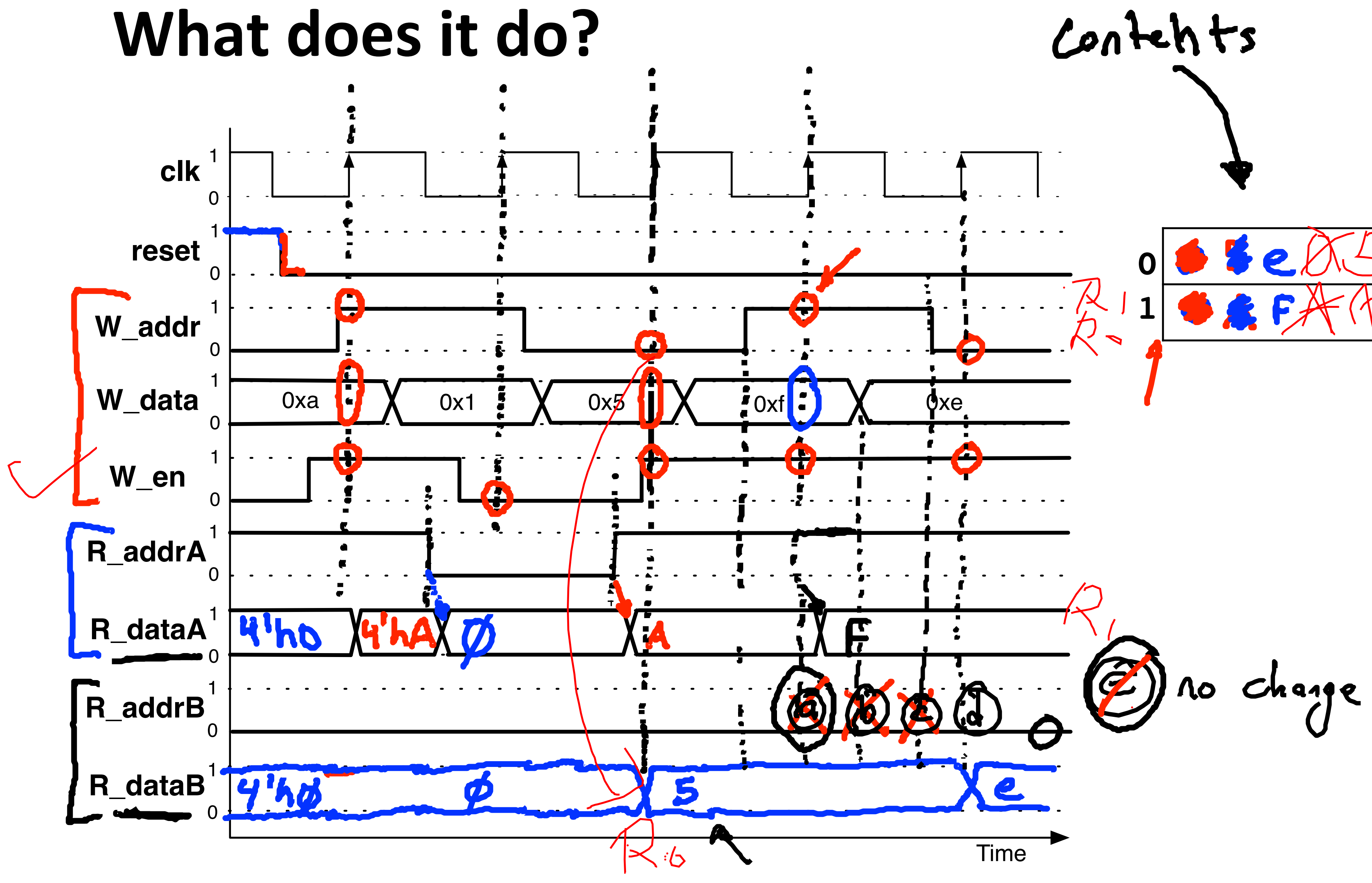
Register file implementation

- A register file has 3 parts
  - The Storage: An array of registers
  - The Read Ports: Output the value of selected register
  - The Write Port: Selectively write one of the registers

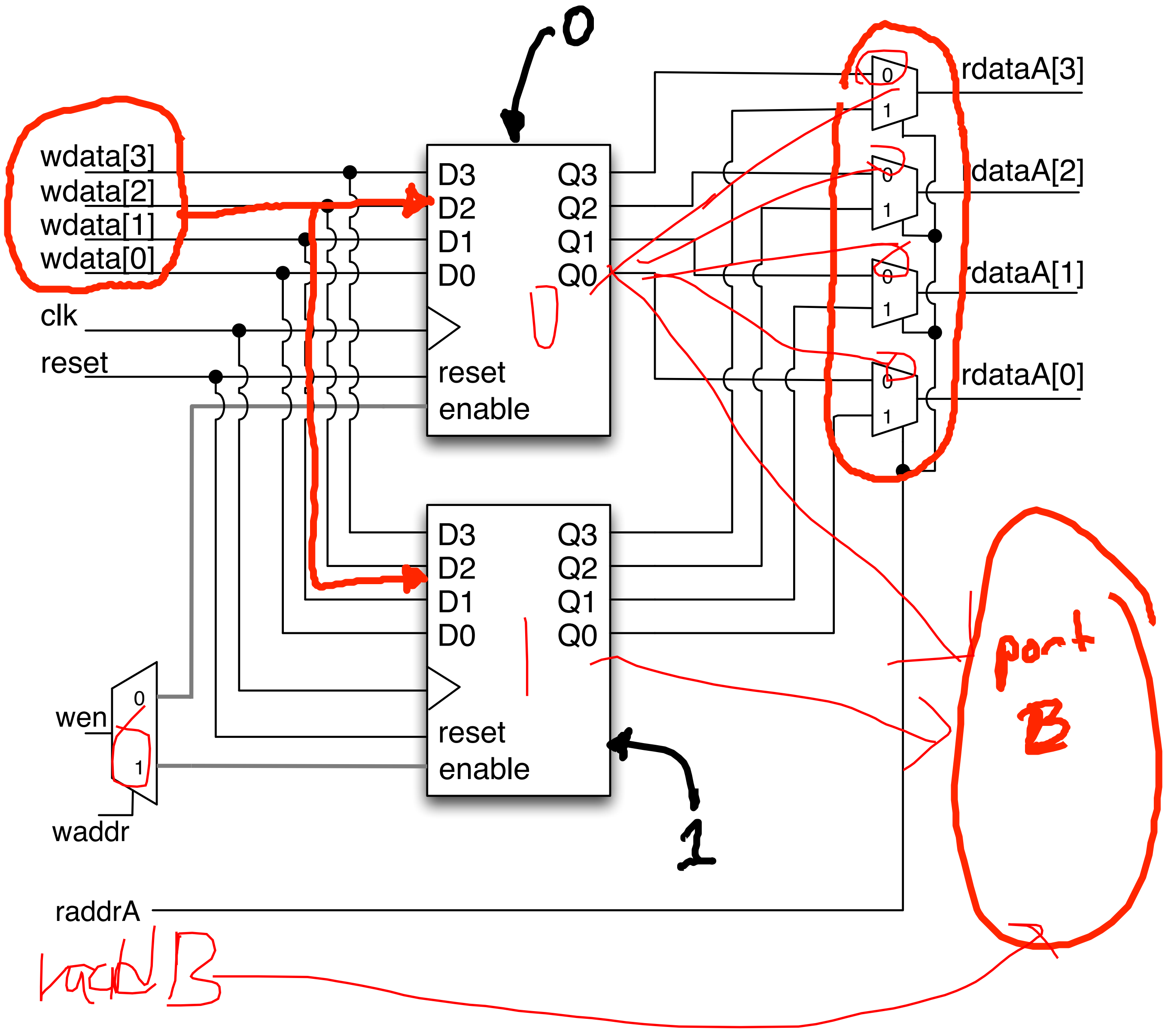


- Let's consider a 2 word memory
  - with 4-bit words

What does it do?

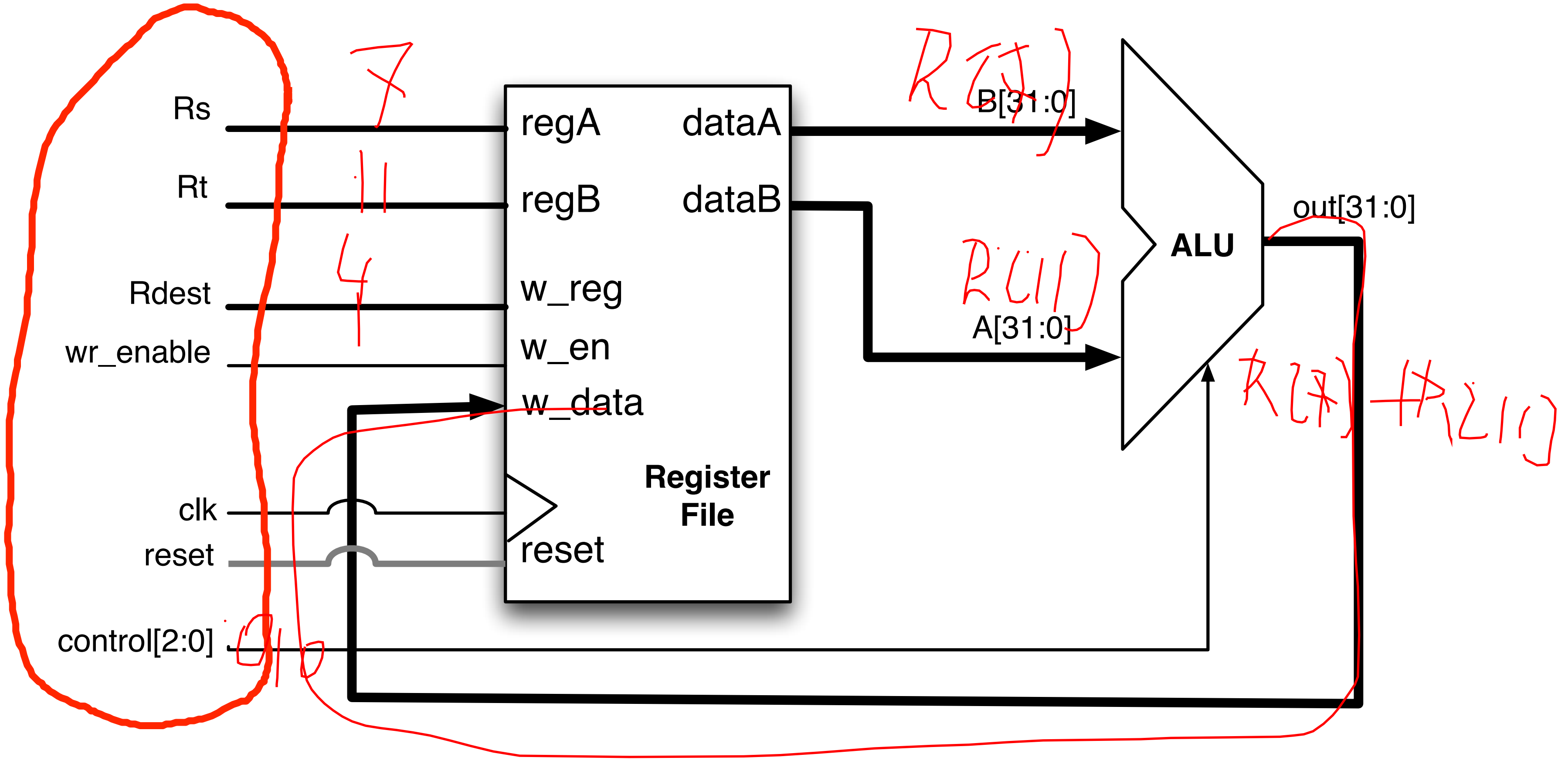


2 x 4-bit register file (only 1 read port shown)



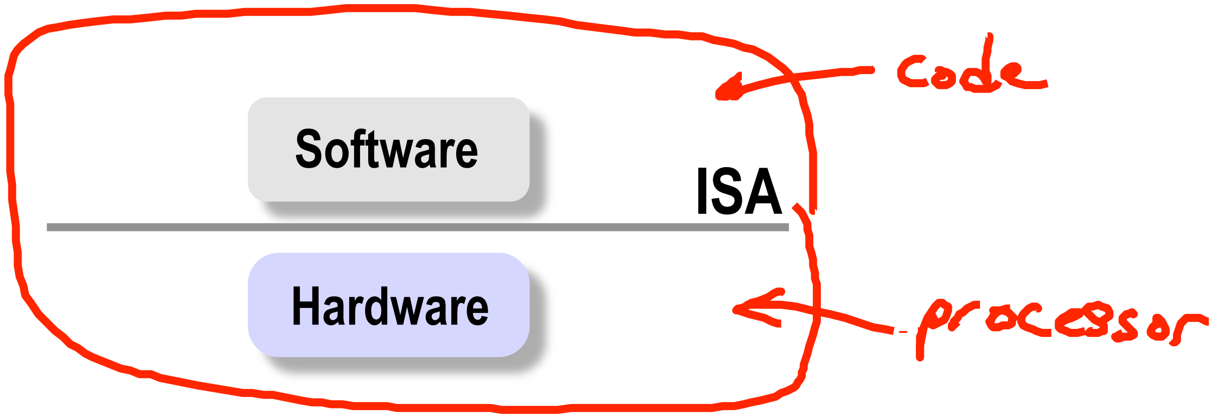
Building an "arithmetic machine"

- **With an ALU and a register file, we can build a calculator**
  - Here are the essential parts.



Building a computer processor.

- The key feature that distinguishes a computer processor from other digital systems is programmability.
- A processor is a hardware system controlled by software



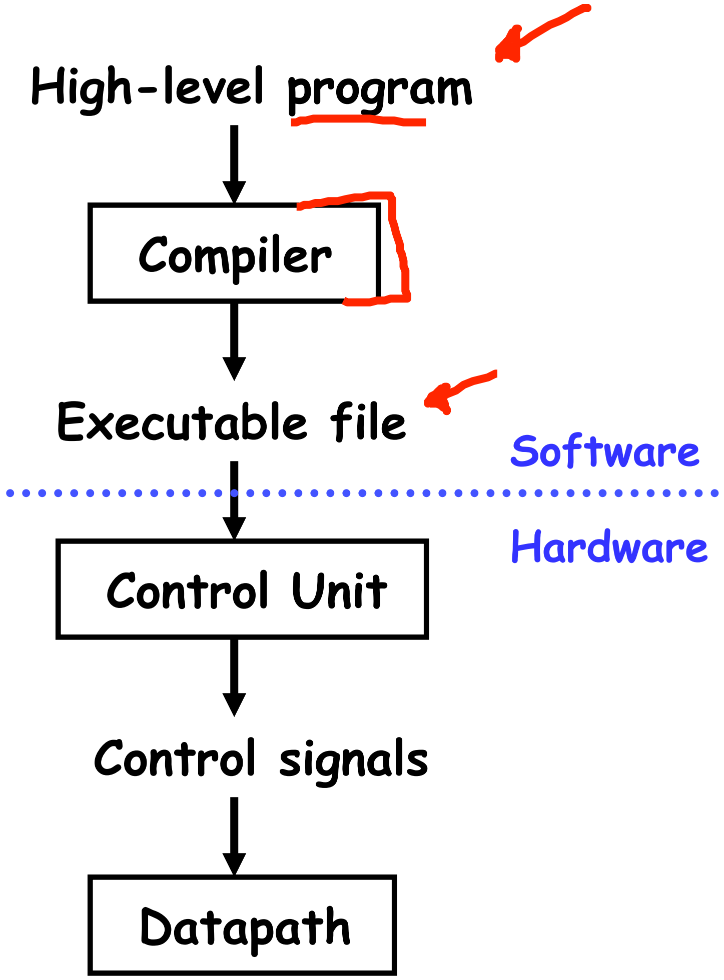
- An Instruction Set Architecture (ISA) describes the interface between the software and the hardware.
  - Specifies what operations are available
  - Specifies the effects of each operation

A MIPS ISA processor

- **Different processor families (x86, PowerPC, ARM, MIPS, …) use their own instruction set architectures.**
- **The processor we'll build will execute a subset of the MIPS ISA**
  - Of course, the concepts are not MIPS-specific
  - MIPS is just convenient because it is real, yet simple
- **The MIPS ISA is widely used. Primarily in embedded systems:**
  - Various routers from Cisco
  - Game machines like the Nintendo 64 and Sony Playstation 2

Programming and CPUs

- Programs written in a high-level language like C++ must be compiled to produce an executable program.
- The result is a CPU-specific machine language program. This can be loaded into memory and executed by the processor.
- Machine language serves as the interface between hardware and software.



High-level languages vs. machine language

- **High-level languages are designed for human usage:**
  - Useful programming constructs (for loops, if/else)
  - Functions for code abstraction; variables for naming data
  - Safety features: type checking, garbage collection
  - Portable across platforms
- **Machine language is designed for efficient hardware implementation**
  - Consists of very simple statements, called **instructions**
  - Data is named by where it is being stored
  - Loops, if/else implemented by branch and jump instructions
  - Little error checking provided; no portability

Assembly Language & Instructions

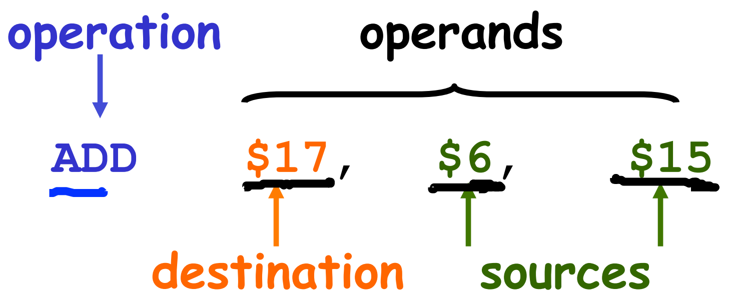
- **Machine language is a binary representation of instructions**
- **Assembly language is a human-readable version**
- **There is an (almost) one-to-one correspondence between assembly and machine languages; we'll see the relation later.**

- **Instructions consist of:**
  - Operation code (*opcode*): names the operation to perform
  - Operands: names the data to operate on
- **Example:**

$$\overset{\text{operation}}{\overset{\downarrow}{\texttt{ADD}}}\quad \overbrace{\texttt{\$17,}\quad \texttt{\$6,}\quad \texttt{\$15}}^{\text{operands}}$$

MIPS: register-to-register, “three address”

- MIPS uses three-address instructions for arithmetic.
  - Each ALU instruction contains a destination and two sources.
- MIPS is a register-to-register architecture.
  - For arithmetic instructions, the destination and sources must all be registers (or constants).
  - Special instructions move values between the register file and memory.
- For example, an addition (a = b + c) might look like:



$6 = register #6

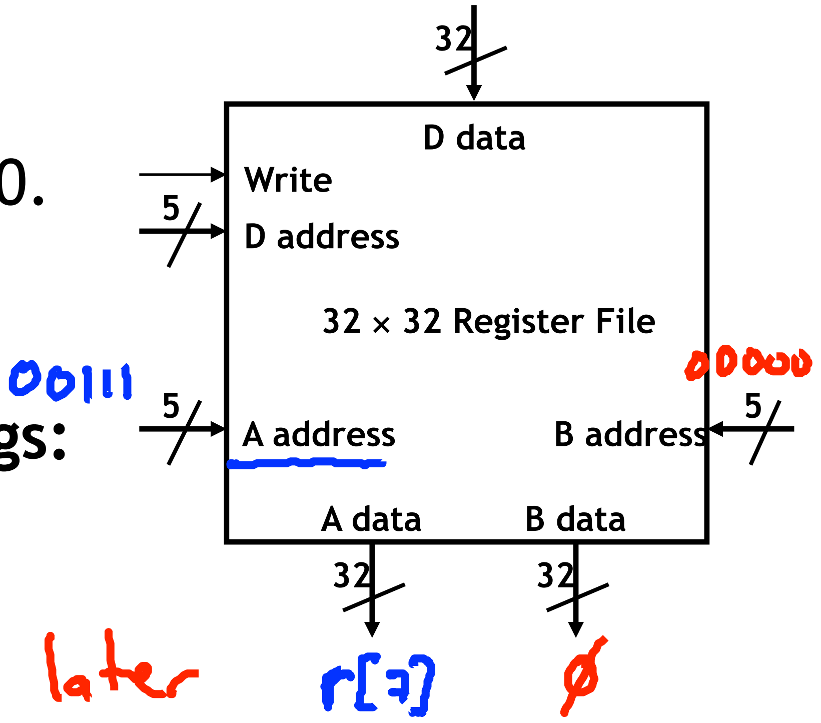
MIPS register file

- MIPS processors have 32 registers, each of which holds a 32-bit value.
  - Register specifiers are 5 bits long.
  - The data inputs and outputs are 32-bits wide.

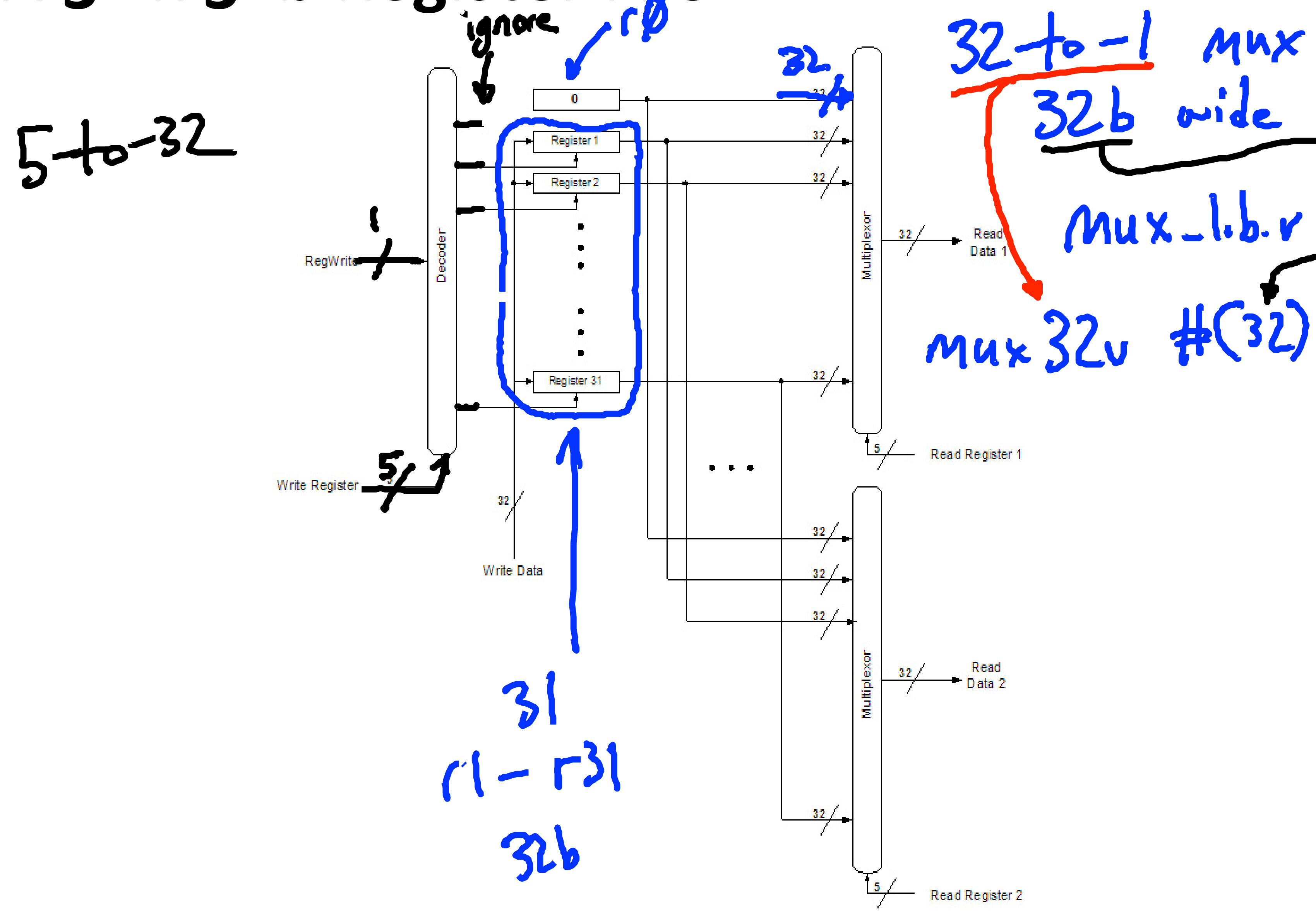
- Register 0 is special
  - It is always read as the value 0.
  - Writes to it are ignored.

- Two naming conventions for regs:
  - By number: \$0,..., \$17,..., \$31
  - By name: \$zero,..., \$s1,..., \$ra

more on this later

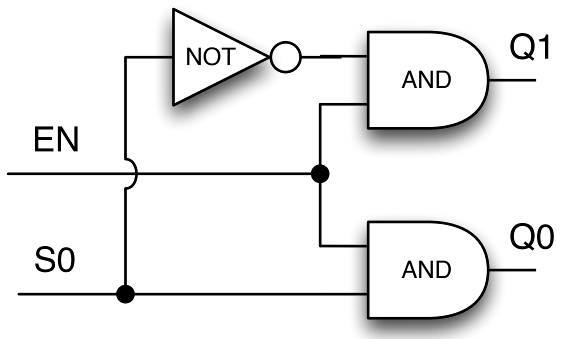


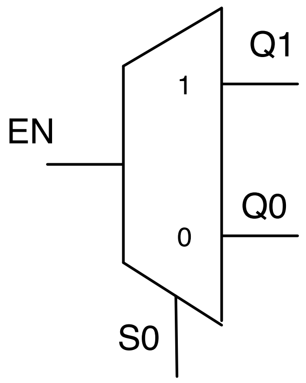
A 32 x 32b Register File



Decoders

- **This circuit is a 1-to-2 decoder**
  - It decodes a 1-bit address, setting the specified output to 1
    - Assuming the circuit is enabled

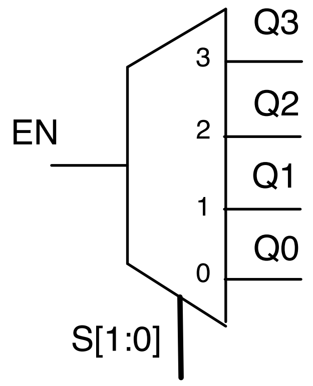




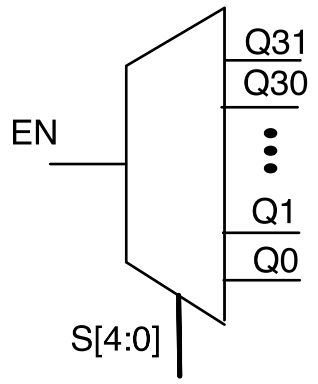
EN	S0	(Q1,Q0)
0	X	(0, 0)
1	0	(0, 1)
1	1	(1, 0)

Scaling Decoders

- **Decoders can be generalized as follows**
- **A 1-to-2^n decoder:**
  - Has a 1-bit enable input, and an n-bit select input
  - Has 2^n outputs
  - All the outputs are zero, except the selectth if enable = 1
    - If enable = 0, all outputs are zero.
- **A 1-to-4 decoder:**



EN	S[1:0]	Q[3:0]
0	X	(0,0,0,0)
1	(0,0)	(0,0,0,1)
1	(0,1)	(0,0,1,0)
1	(1,0)	(0,1,0,0)
1	(1,1)	(1,0,0,0)



EN	S[4:0]	Q[31:0]
0	X	0x0000
1	0	0x0001
1	1	0x0002
1	…	…
1	30	0x4000
1	31	0x8000

Basic arithmetic and logic operations

- MIPS provides basic integer arithmetic operations:

  add sub mul* div*

- And logical operations:

  and or nor xor not

- Remember that these all require three register operands; for example:

```
add  $14, $18, $3      # $14 = $18 + $3
mul  $22, $22, $11     # $22 = $22 x $11
```

Note: a full MIPS ISA reference can be found in Appendix A (linked from website)

** We won't implement these in our implementation*

Larger expressions

- More complex arithmetic expressions may require multiple operations at the instruction level.

$$\$4 = (\underbrace{\underline{\$1 + \$2}) \times (\underline{\$3 - \$4}})$$

```
add  $5, $1, $2        # $5 contains $1 + $2
sub  $4, $3, $4        # Temporary value $4 = $3 - $4
mul  $4, $4, $5        # $4 contains the final product
```

- Temporary registers may be necessary, since each MIPS instructions can access only two source registers and one destination.
  - could have re-used $1 instead of introducing $5.
  - But be careful not to modify registers that are needed again later.

Immediate operands = Constants

- **So far, the instructions expect register operands. How do you get data into registers in the first place?**
  - Some instructions allow you to specify a signed constant, or "immediate" value, for the second source instead of a register.
  - For example, here is the immediate add instruction, addi:

```
addi $15, $1, 4      # $15 = $1 + 4
```

  - Immediate operands can be used in conjunction with the $zero register to write constants into registers:

```
addi $15, $0, 4      # $15 = 4
```

A more complete example

- What if we wanted to compute the following?

$$1 + 2 + 3 + 4$$

A more complete example

- What if we wanted to compute the following?

must be register

$$1 + 2 + 3 + 4$$

I
```
addi $1, 1, 2
addi $2, 3, 4
add  $1, $1, $2
```

II
```
addi $1, $0, 1
addi $1, $1, 2
addi $1, $1, 3
addi $1, $1, 4
```

III
```
addi $1, $0, 1
addi $2, $0, 2
addi $3, $0, 3
addi $4, $0, 4
add  $1, $1, $2
add  $3, $3, $4
add  $1, $1, $3
```

A: none of the above
B: I and II
C: I and III
[✓ D: II and III]
E: all of the above