Keep MIPS Reference Sheet

Building an Arithmetic Machine

Exam 1

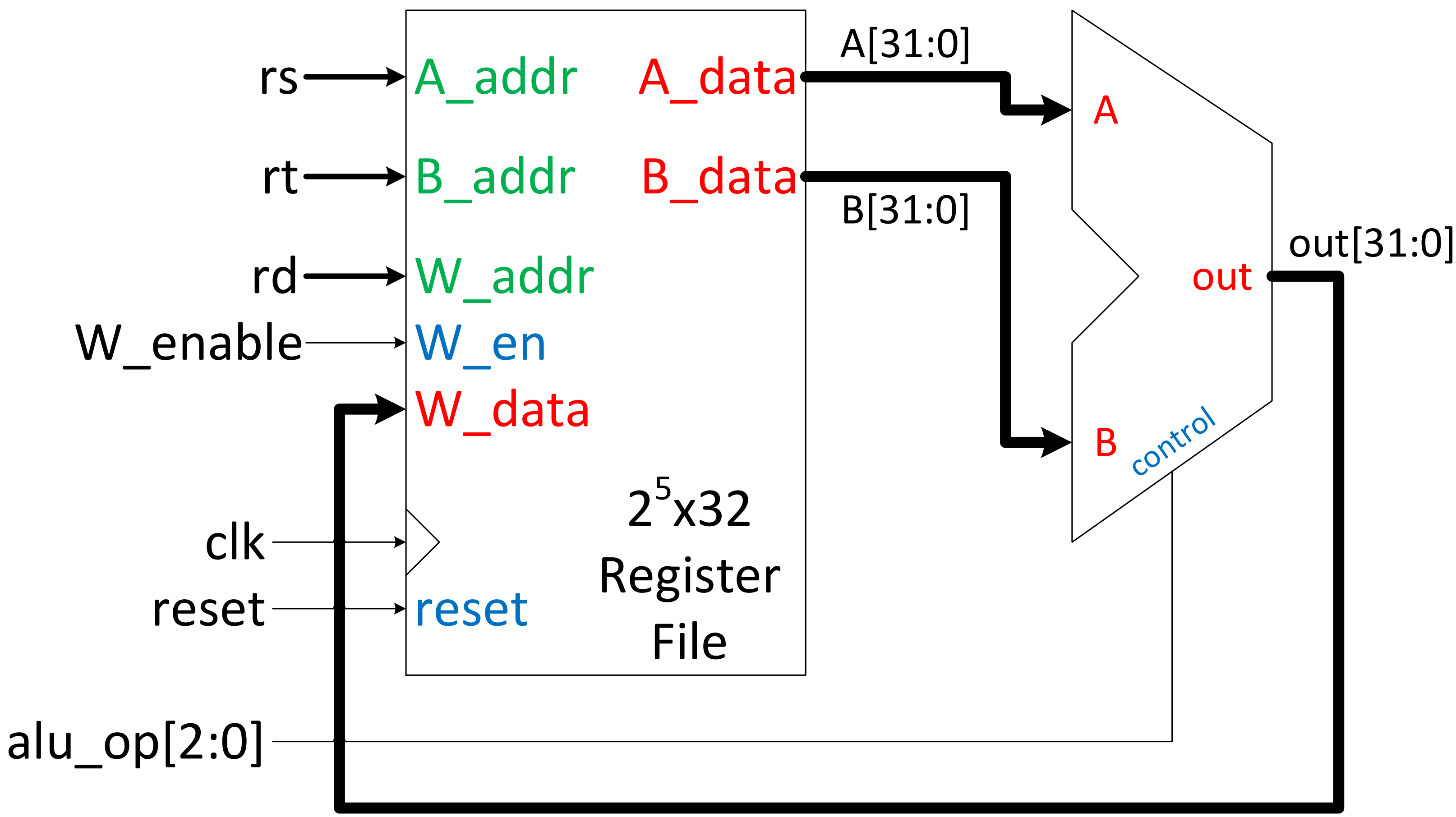
Study sessions (MWF, 2-3pm)

2 Handouts

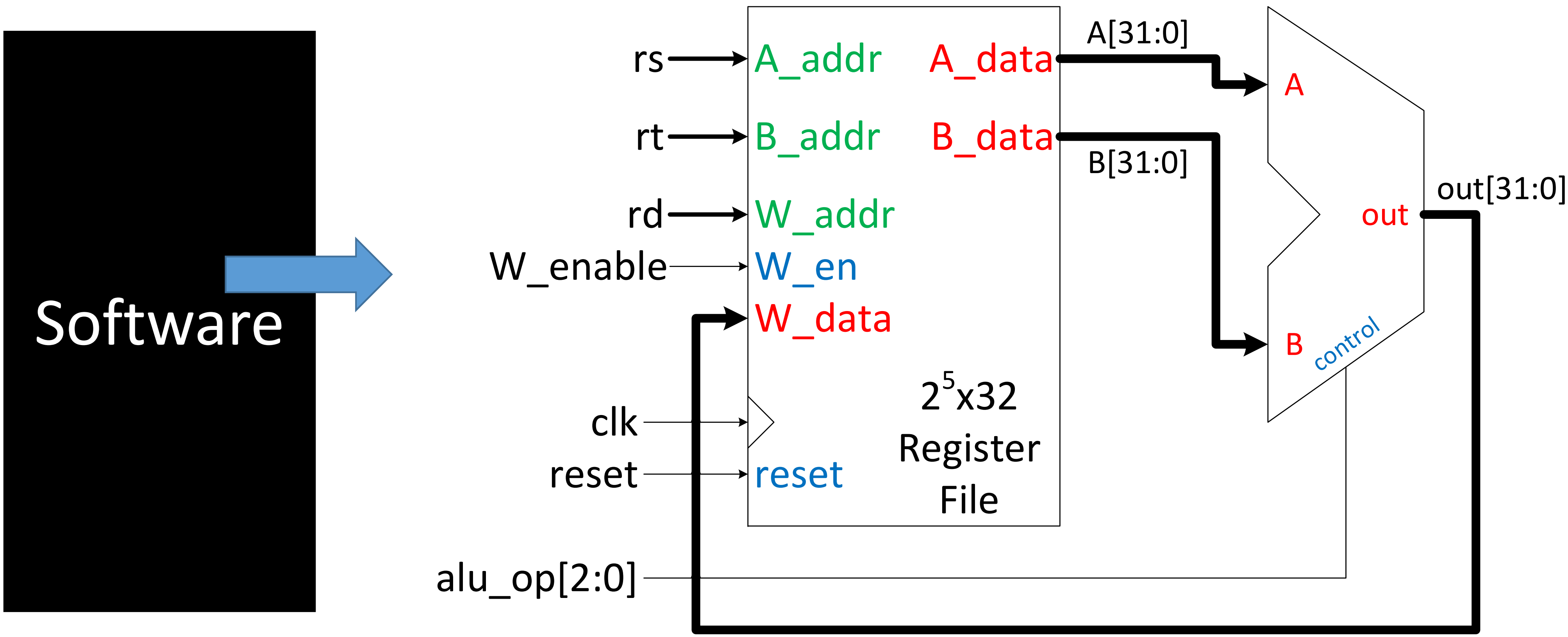
Today's lecture

- The Arithmetic Machine
  - Programmable hardware
  - Instruction Set Architectures (ISA)
  - Instructions & Registers
    - Assembly Language
    - Machine Language
- Storing and manipulating state on the arithmetic machine

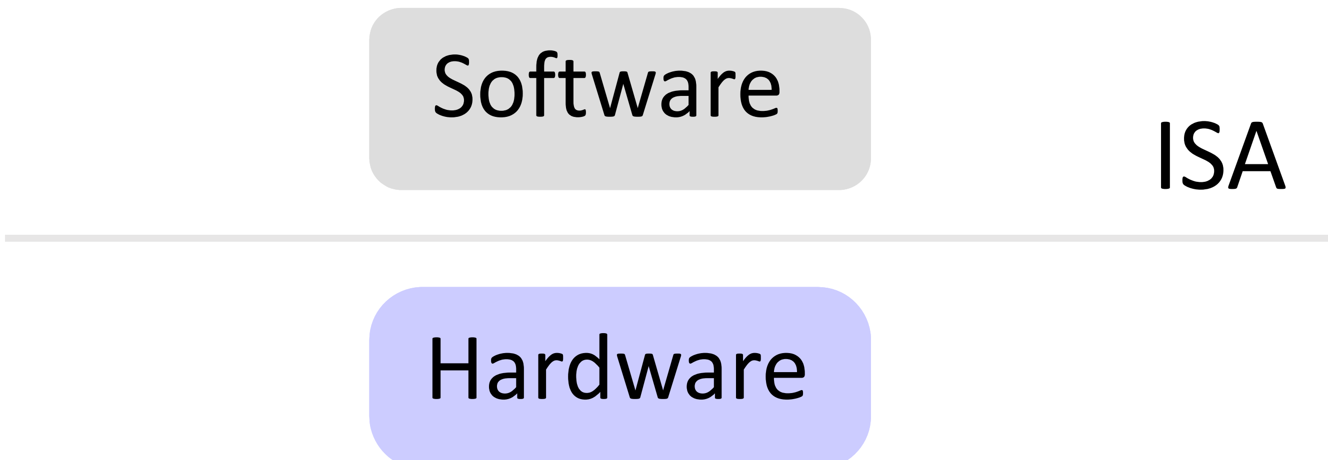
With an ALU and a register file, we can build an “arithmetic machine”



A processor is different from other datapaths because it is programmable



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

ISAs describe families of processors

x86 and x64

ARM

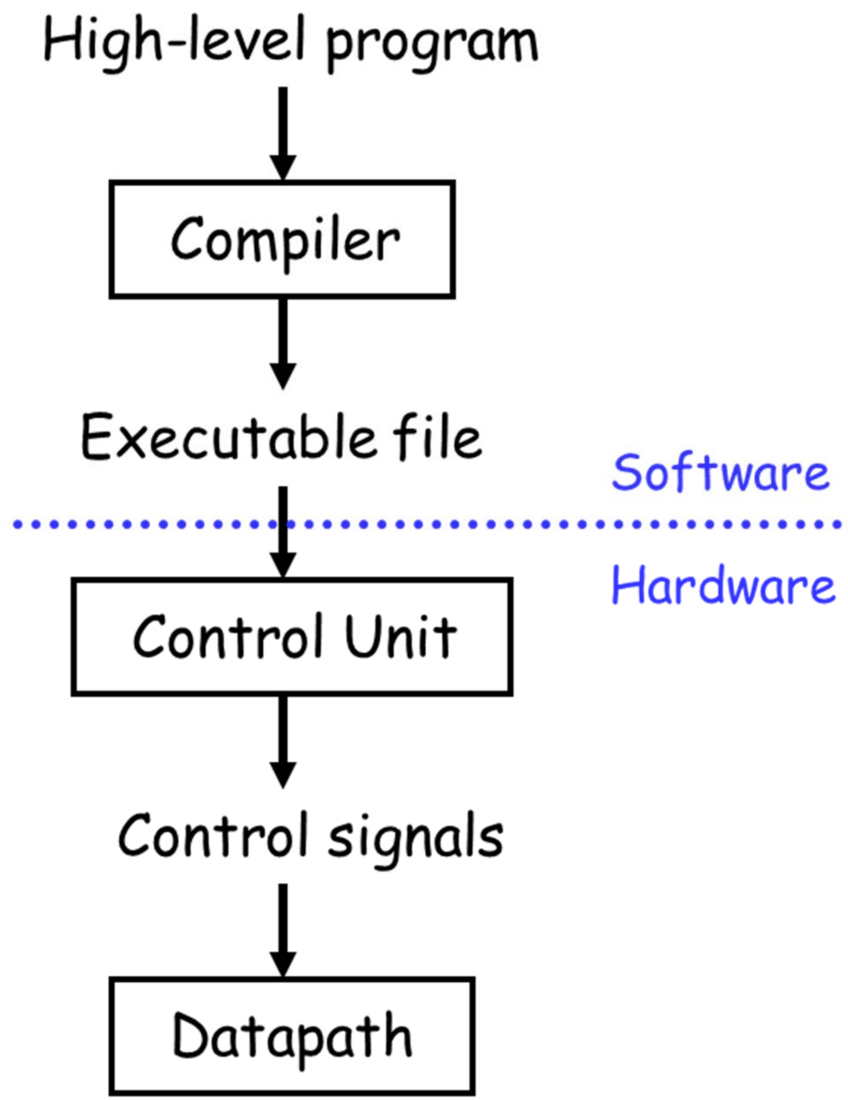
MIPS

We will teach a subset of MIPS

- Concepts we teach transcend MIPS
- MIPS is real, yet simple
- The MIPS ISA is primarily used in embedded systems:

High-level languages get compiled to ISA-specific executable programs

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 is a human readable version of binary machine languages

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

MIPS is a register-to-register architecture: Arithmetic/logical state manipulations read from registers (or constants) and write to registers

- Each ALU instruction contains a destination and two sources.
- Special instructions move state information between the register file and main memory.
- For example, an addition (a = b + c) might look like:

operation

operands

rd rs rt

ADD $17, $6, $15

destination sources

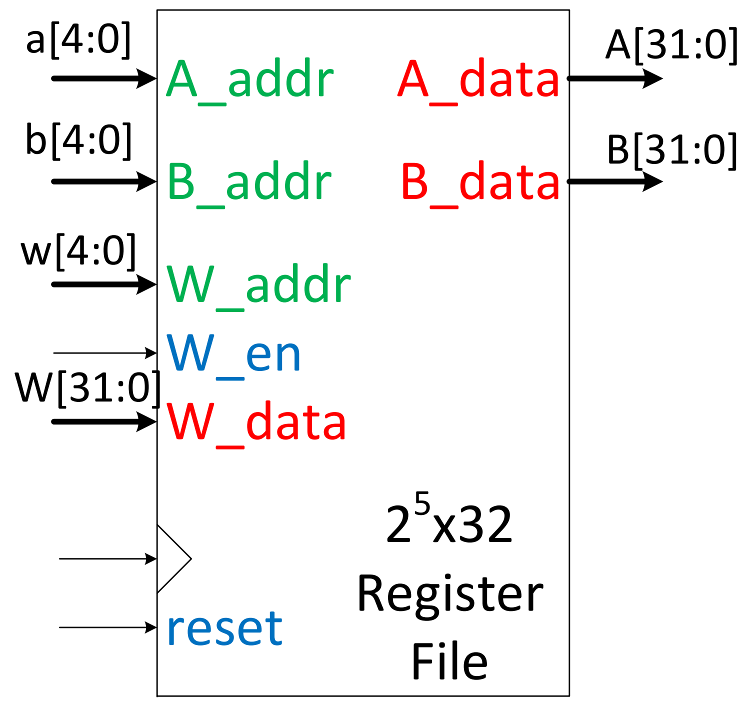
$6 = register #6

MIPS register file has 32 registers, each hold a 32-bit value

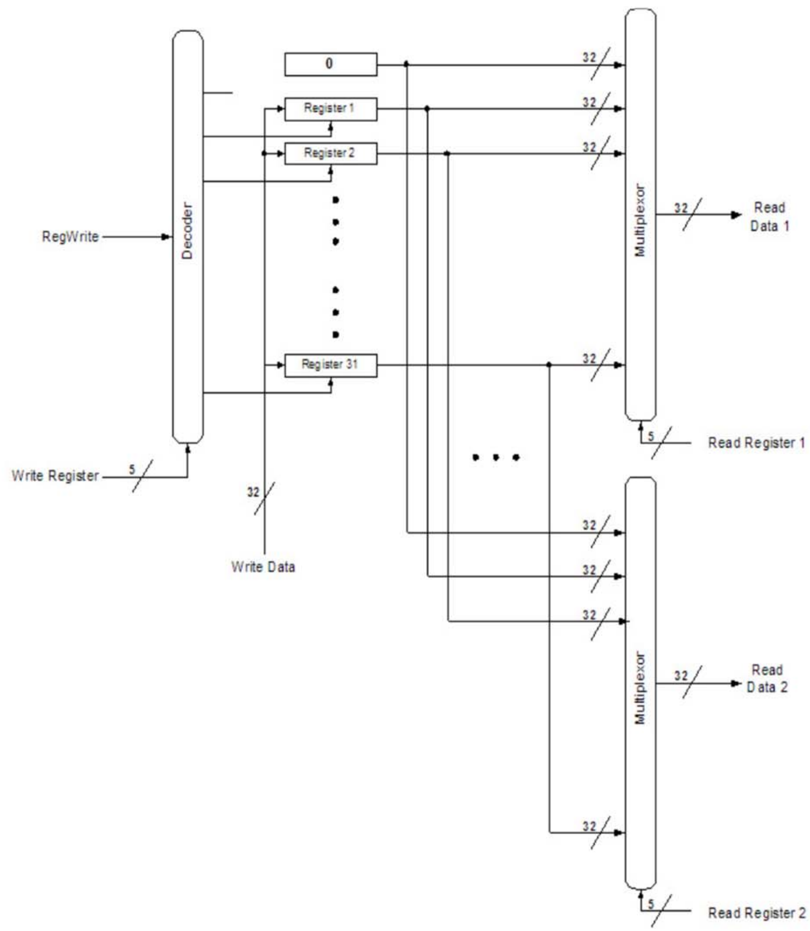
- Register specifiers (addresses) are 5 bits long.
- The data inputs and outputs are 32-bits wide.

- 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



A 32 x 32b Register File



MIPS supports basic arithmetic and logical instructions

- Arithmetic operations:

  add   sub   mul*   div*

- 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*

A computer does 2 things: store state and manipulate state

(1)

MIPS Reference Data

CORE INSTRUCTION

New state to store

Stored State

NAME, MNEMONIC		MAT	OPERATION (in Verilog)
Add	add	R	R[rd] = R[rs] + R[rt]
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]
And	and	R	R[rd] = R[rs] & R[rt]

State manipulation

Quick aside on arrays: What’s the difference?

```
char *string;

string[i]  *(string+i)
```

Arrays use “base + offset” addressing

```
char *string;
```

Base

Offset

```
string[3]

*(string+3)
```

Address	Data
0	NULL
1	NULL
2	‘C’
3	‘S’
4	‘2’
5	‘3’
6	‘3’
7	????
8	????

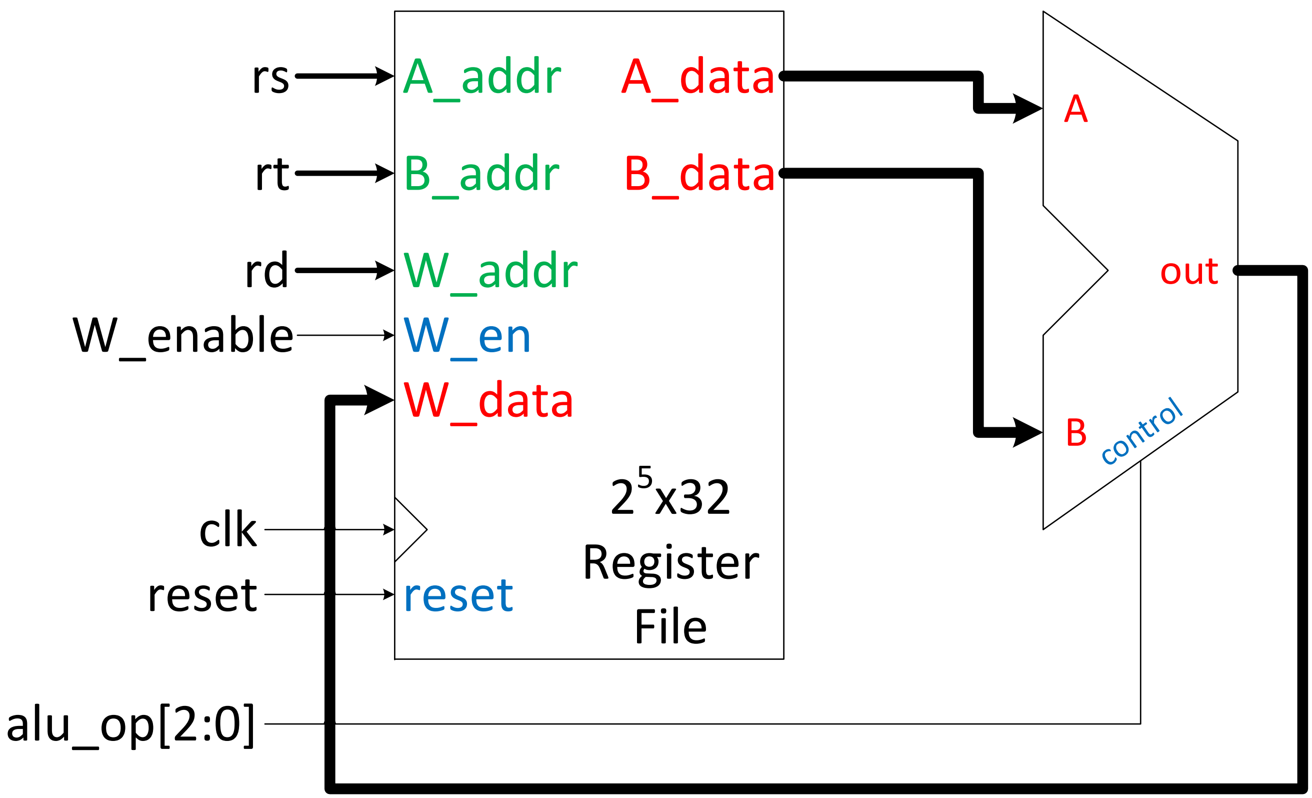
rs, rd, and rt tell us our offset from the “top” of our “register array”

R[3]

rs	Data
0	84
1	6584
2	4248
3	$\boxed{6485}$
4	1388
…	
…	
N-2	841607
N-1	0

Instructions tell us where to find the data we want to manipulate or where to store data

`add   rd, rs, rt` ⟺ `R[rd] = R[rs] + R[rt]`

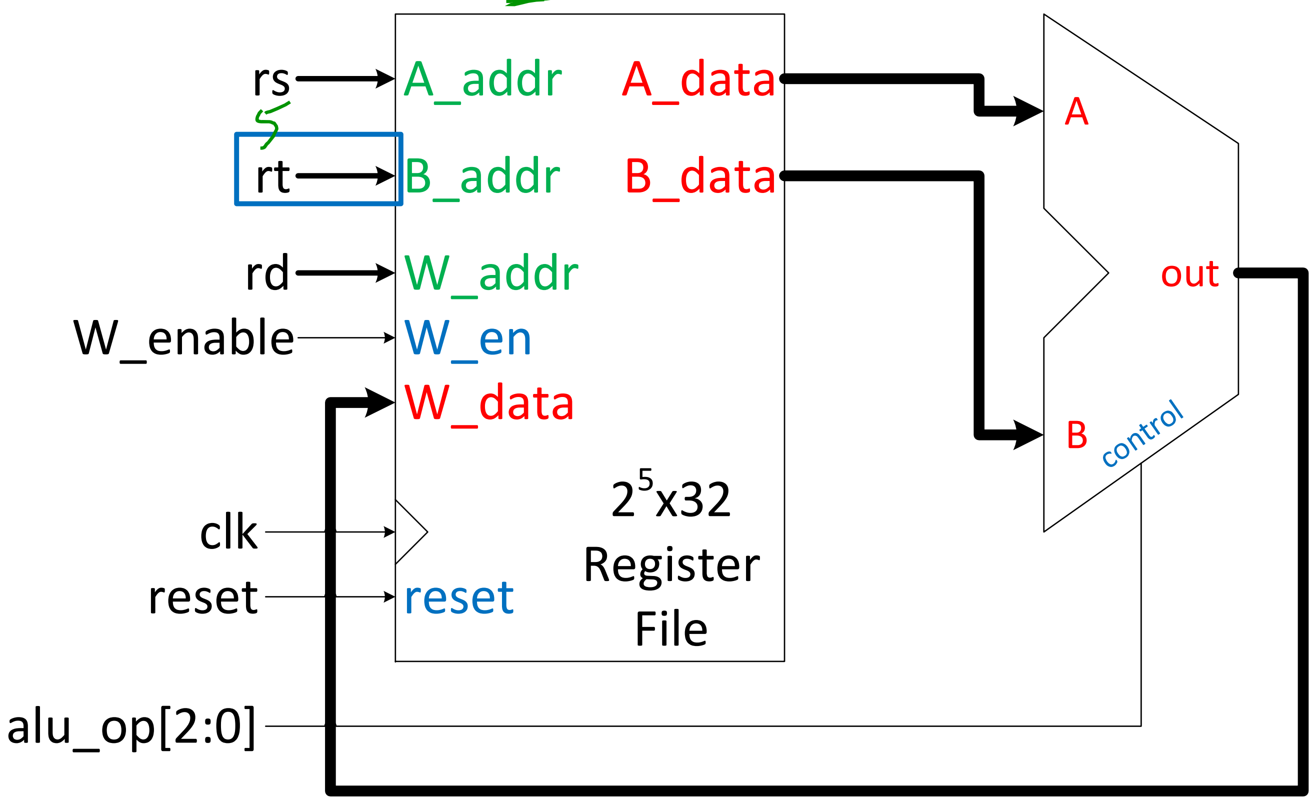


Register File

0	00000000
1	12345678
2	9ABCDEF0
3	DEADBEEF
4	FEEDFACE
5	FFFFFFFF
6	AAAAAAA
7	C0DED00D

i>clicker question

add rd **$7**, rs **$3**, rt **$5**



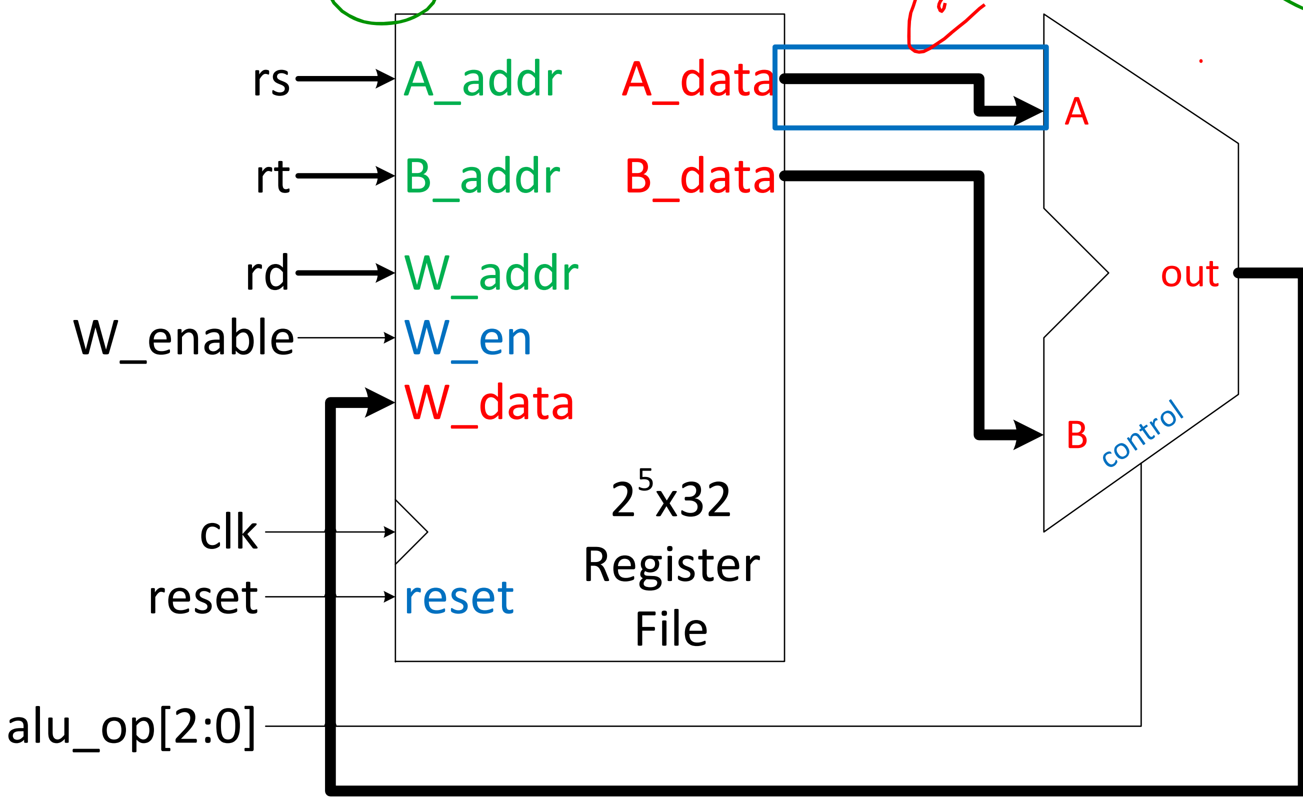
0	00000000
1	
2	
3	0000000A
4	
5	FFFFFFF9
6	
7	00000005

What decimal value is on the bus?

a) -7 b) 3
c) 5 d) 7
e) 10

i>clicker question

add $7, $3, $5



0	00000000
1	
2	
3	0000000A
4	
5	FFFFFFF9
6	
7	00000005

What decimal value is on the bus?

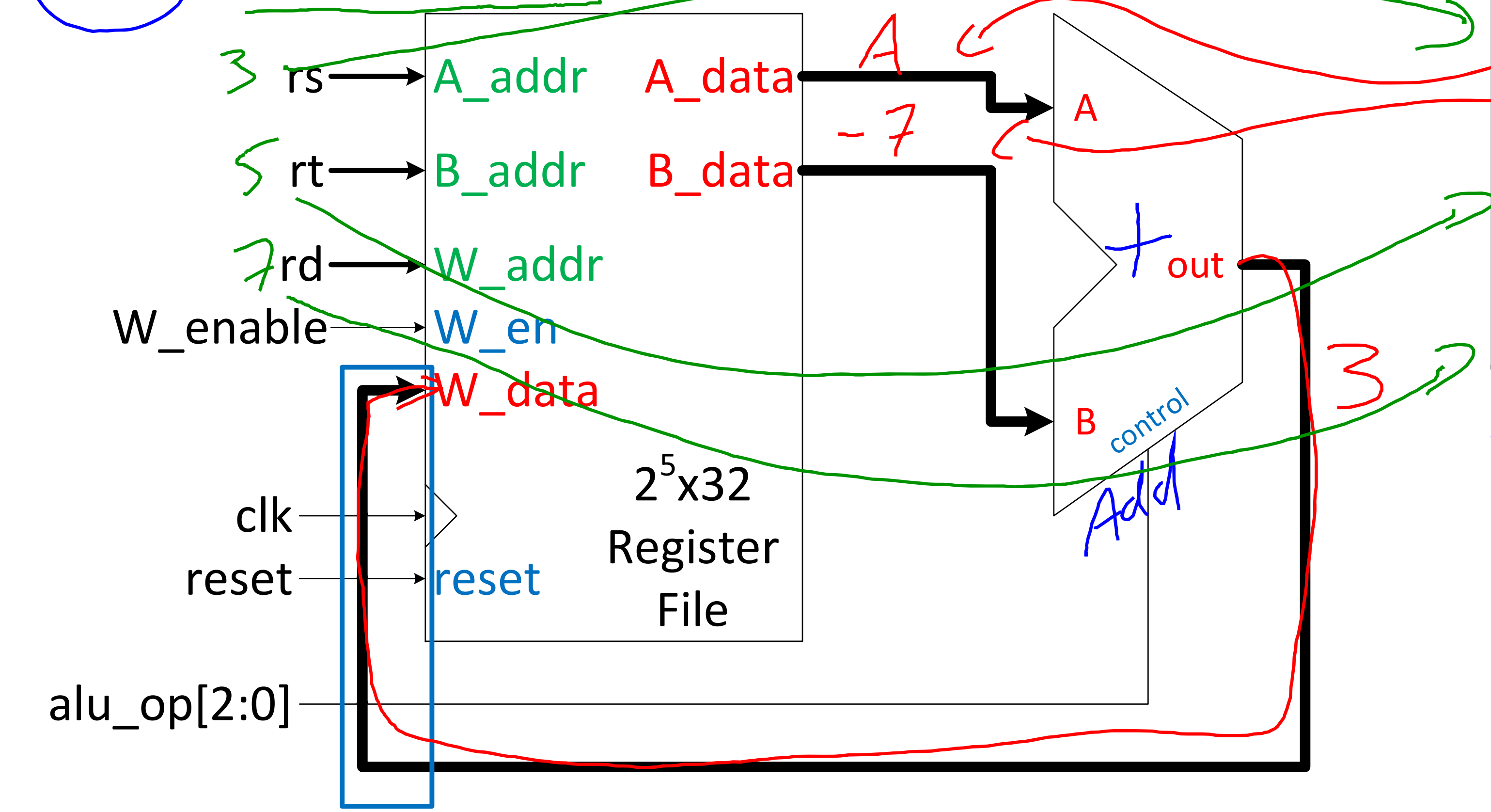
a) -7 b) 3

c) 5 d) 7

e) 10

i>clicker question

rd rs rt
add $7, $3, $5

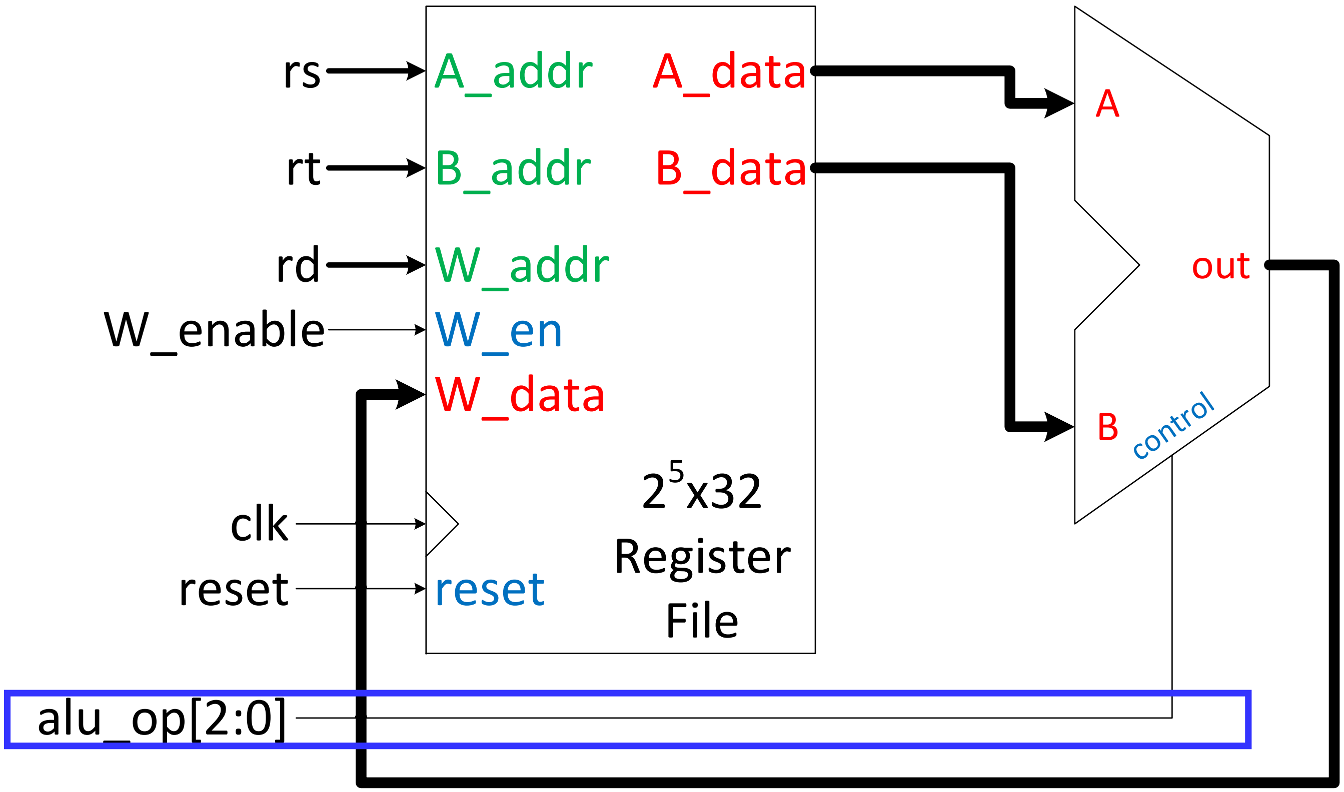


0	00000000
1	
2	
3	0000000A
4	
5	FFFFFFF9
6	
7	~~00000005~~ 3

What decimal value is on the bus?

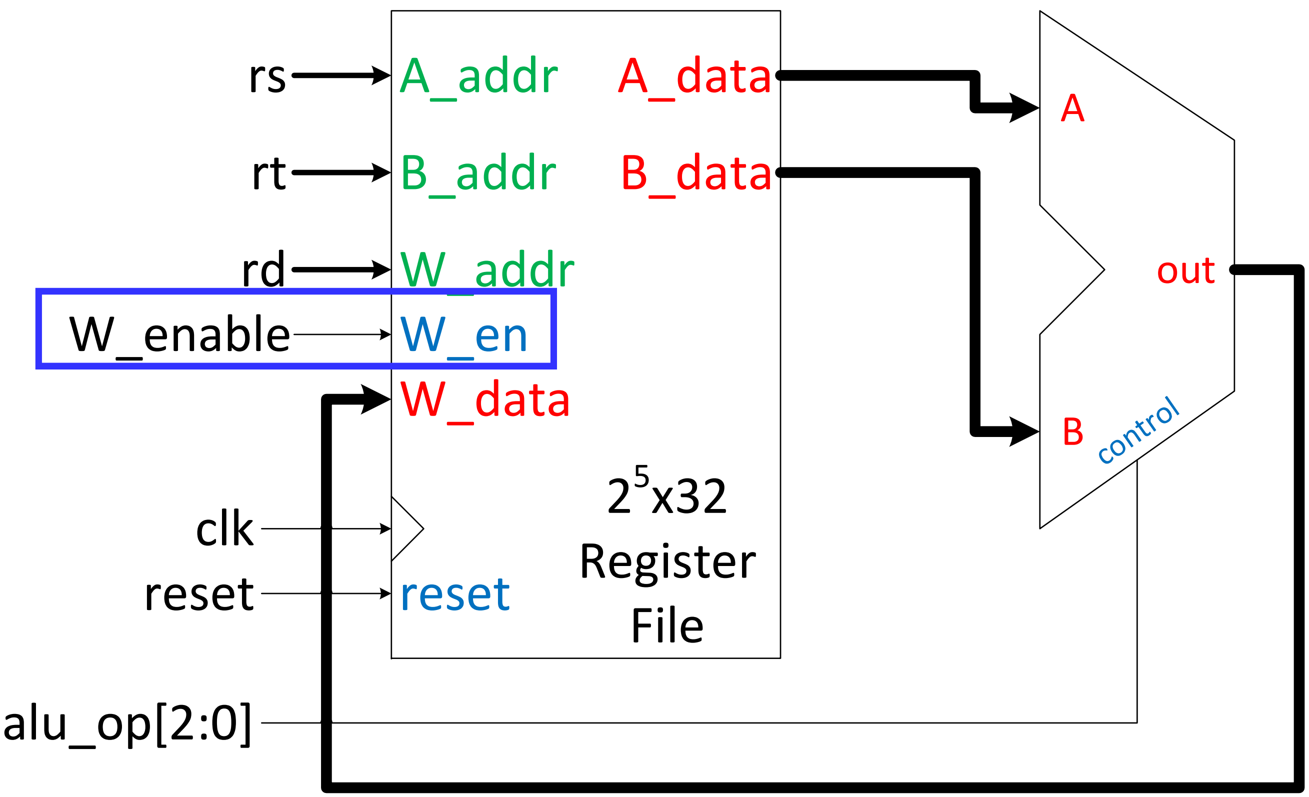
a) -7 b) 3
c) 5 d) 7
e) 10

If we change our control bits, we can change our state manipulations



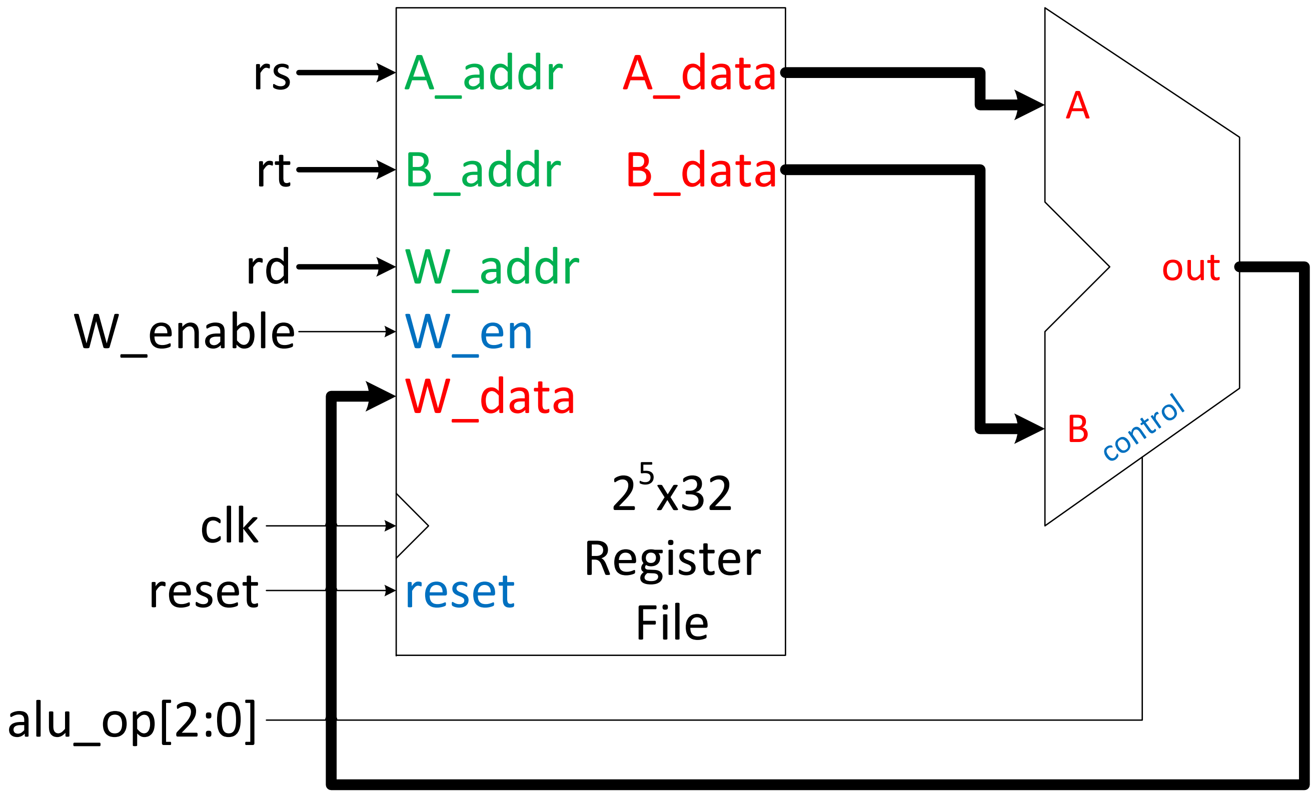
alu_op	Operation
010	ADD
011	SUB
100	AND
101	OR
110	NOR
111	XOR

If we change our control bits, we can change our state manipulations



wr_enable	Operation
0	nop
1	See ALU

Manipulation of the arithmetic machine datapath requires that we correctly differentiate between data, control, and addresses



Immediate operands let the user send data onto the datapath with their instruction

- In MIPS, immediates are always and only the second operator
- Add immediate instruction, addi:

```
addi $15, $1, 4  # R[15] = R[1] + 4
```

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)
Add	add	R	R[rd] = R[rs] + R[rt]
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm

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

rt

```
addi $15, $0, 4  # R[15] = 4
```

+0

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)
Add	add	R	R[rd] = R[rs] + R[rt]
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm

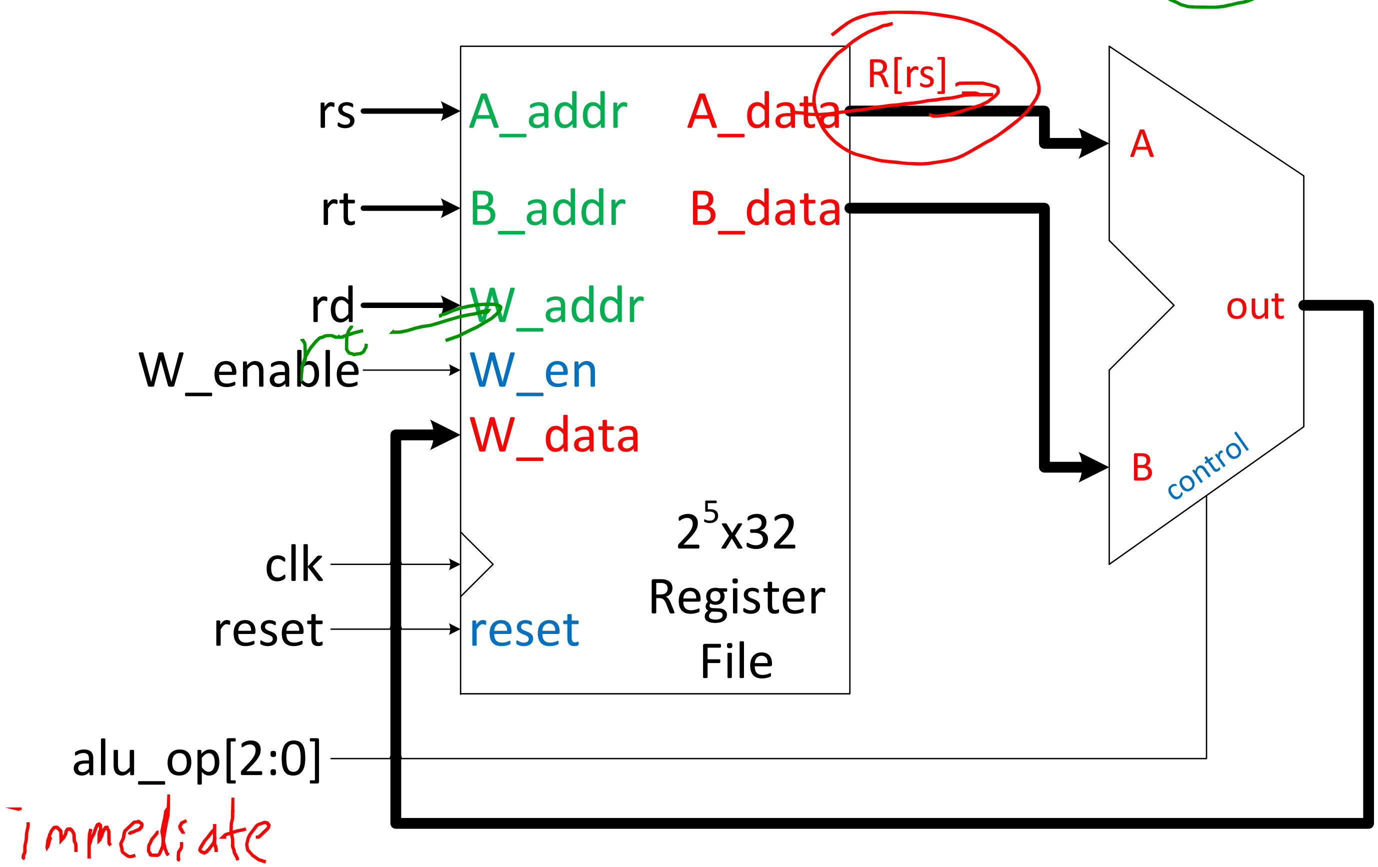
Modify the datapath with three "easy" steps

1) Use verilog to find your state/data sources and destinations
2) Route your data through the component that can perform your desired state manipulations
3) Add multiplexers and their control signals as needed to choose between existing state manipulations and new ones that conflict.

This is the entirety of what we will be doing for Lectures 12-15 and Exam 3!

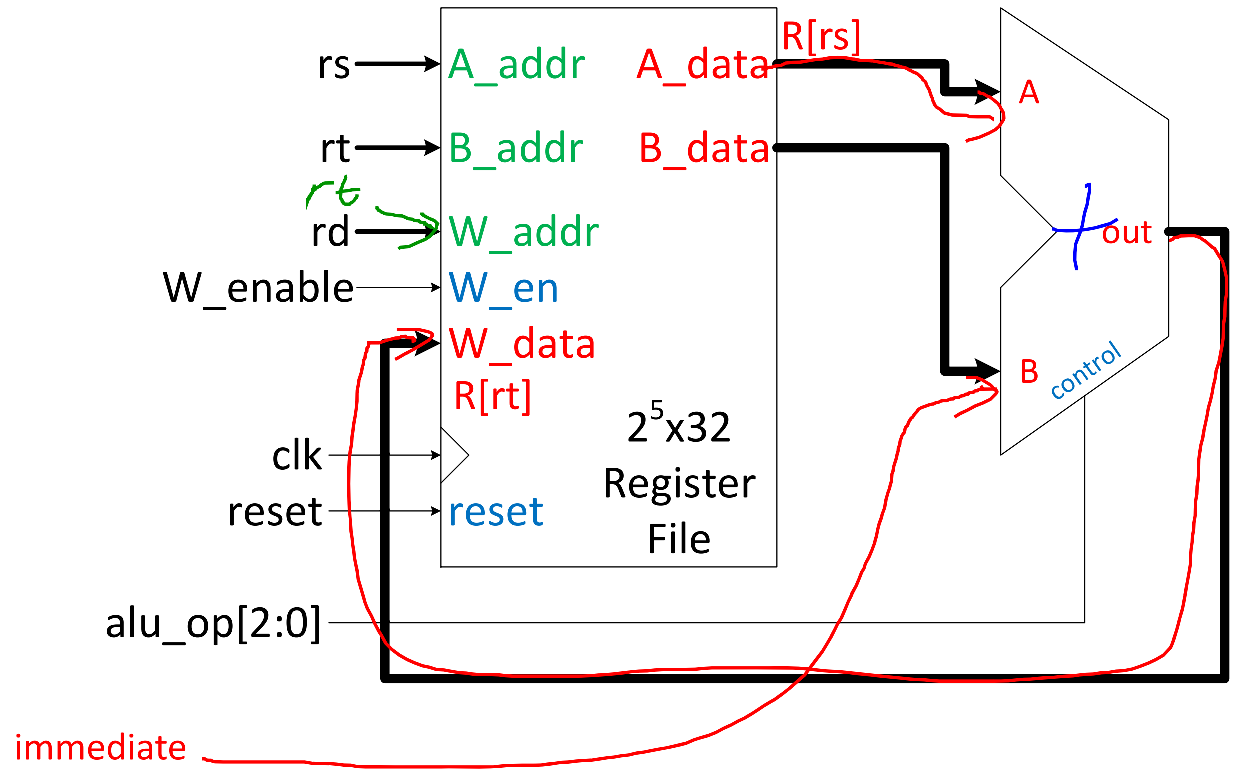
1) Find your data

`addi rt, rs, immediate`   `# R[rt] = R[rs] + immediate`



2) Find a route for your data

`addi rt, rs, immediate          # R[rt] = R[rs] + immediate`



3) Add multiplexers as needed

```
add  rd, rs, rt              # R[rd] = R[rs] + R[rt]
addi rt, rs, immediate       # R[rt] = R[rs] + immediate
```