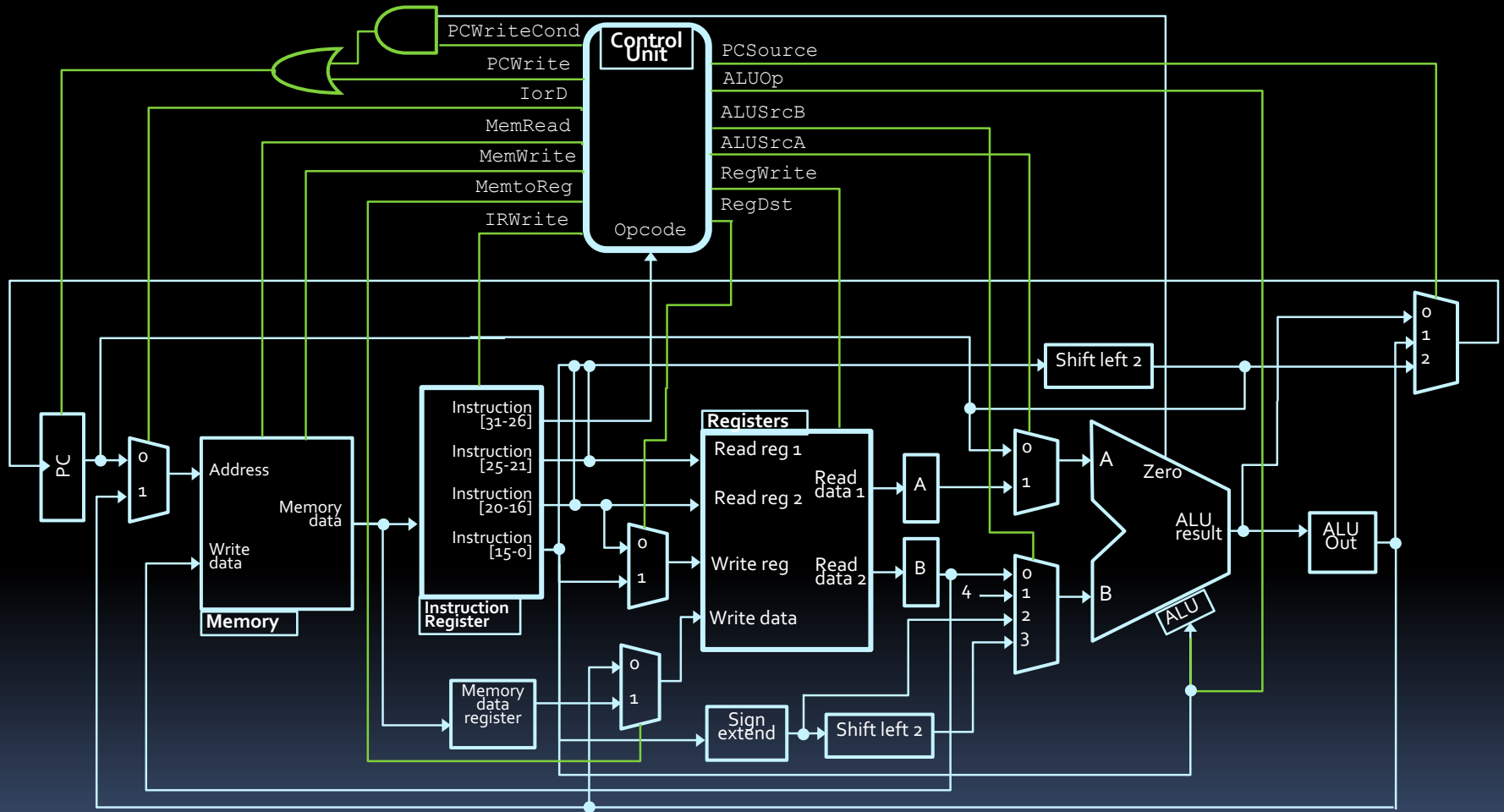
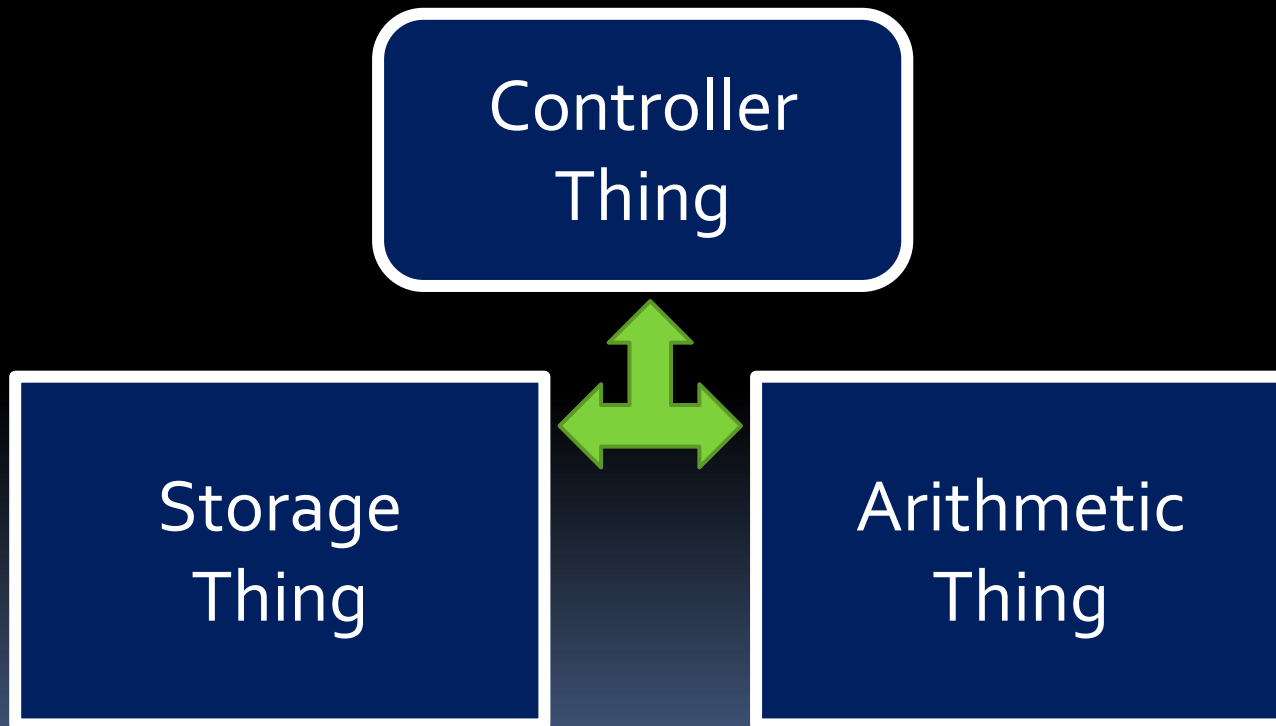


Lecture 7: Processor Storage and Control

The Final Destination



Deconstructing processors



The “Storage Thing”

aka: the register file and main memory



Memory and registers

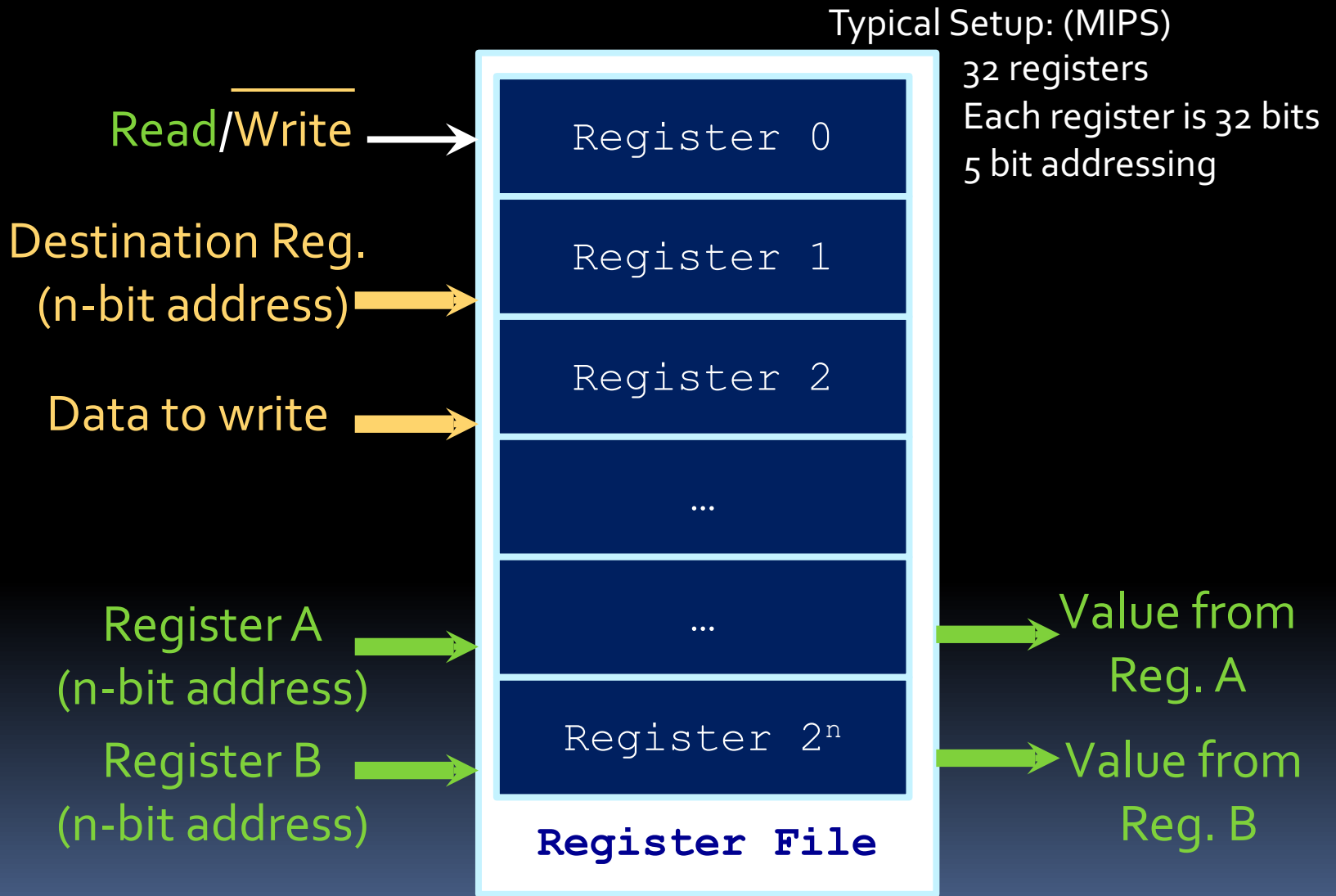
- The processor has registers that store a single value (program counters, instruction registers, etc.)
- There are also units in the CPU that store large amounts of data for use by the CPU:
 - **Register file:** Small number of fast memory units that allow multiple values to be read and written simultaneously.
 - **Main memory:** Larger grid of memory cells that are used to store the main information to be processed by the CPU.

Memory and registers

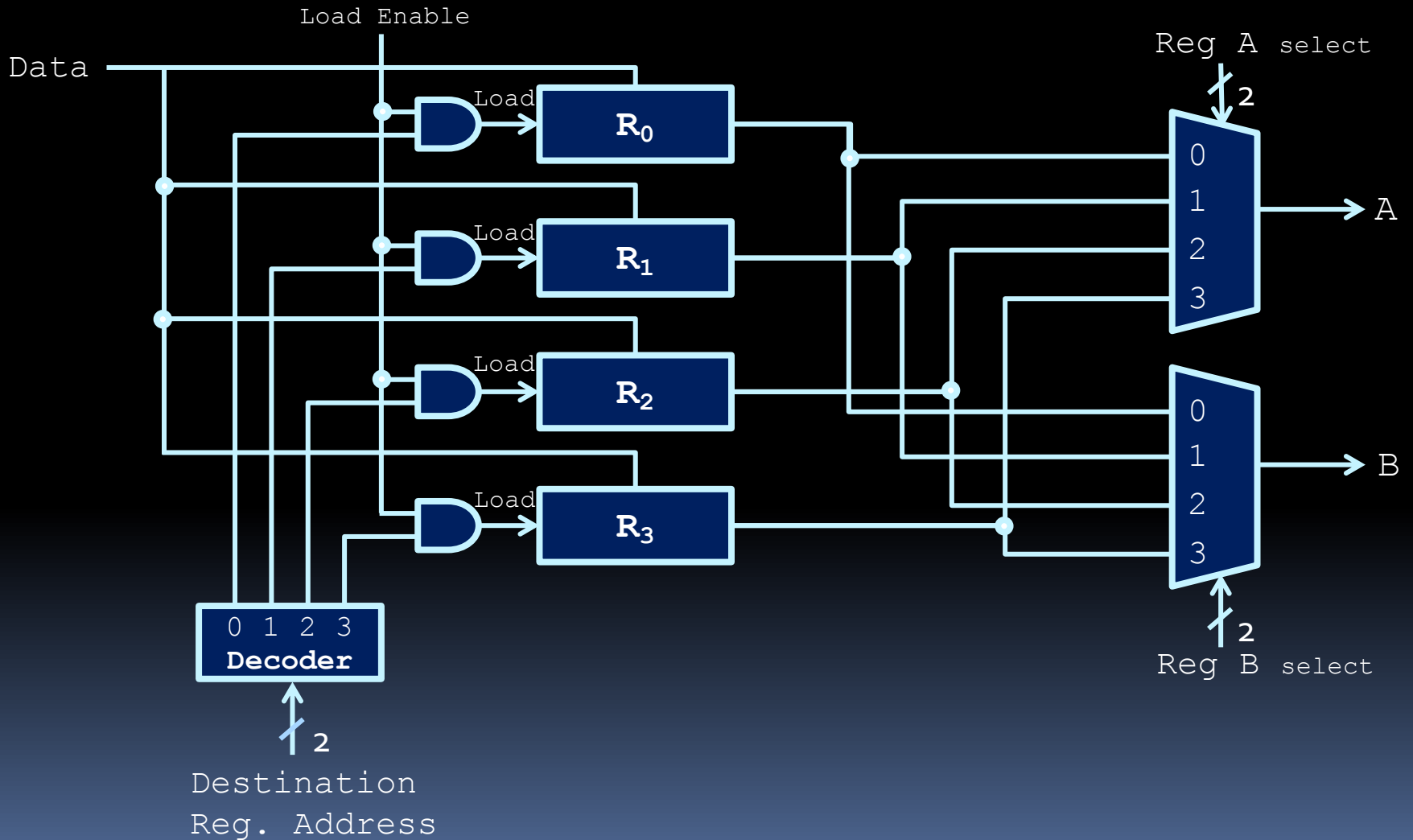
In terms of access speed:

- Register = The plate in front of you
- Cache = The fridge in the kitchen
- Memory (RAM) = The corner grocery store
- Hard Disk = The farm in the prairies
- Network = The farm in another country

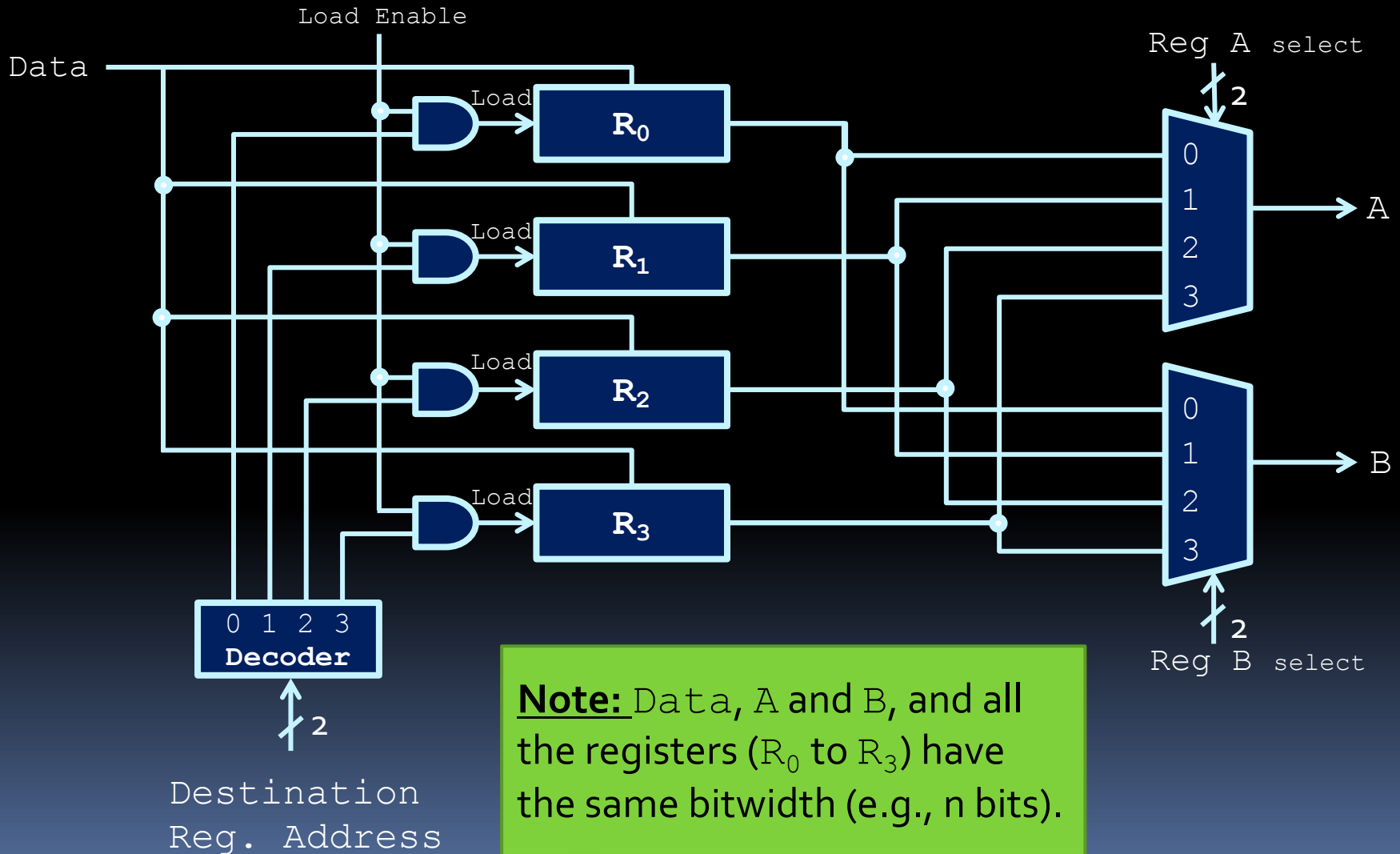
Register File Functionality



Register File - Write Operation



Register File - Read Operation

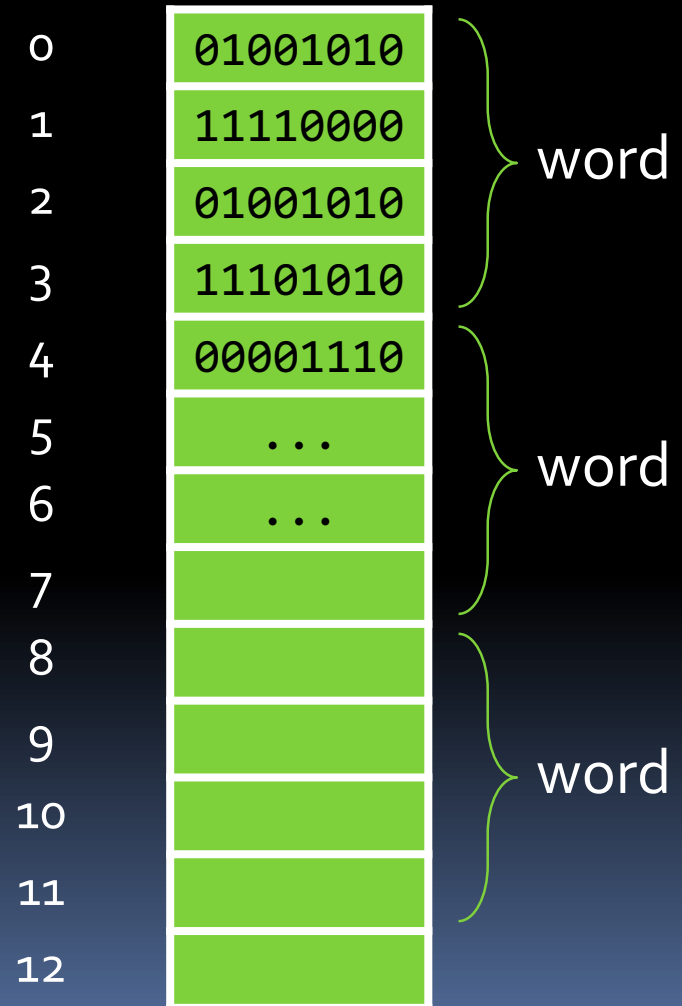


Note: Data, A and B, and all the registers (R_0 to R_3) have the same bitwidth (e.g., n bits).

Main Memory and Addressing

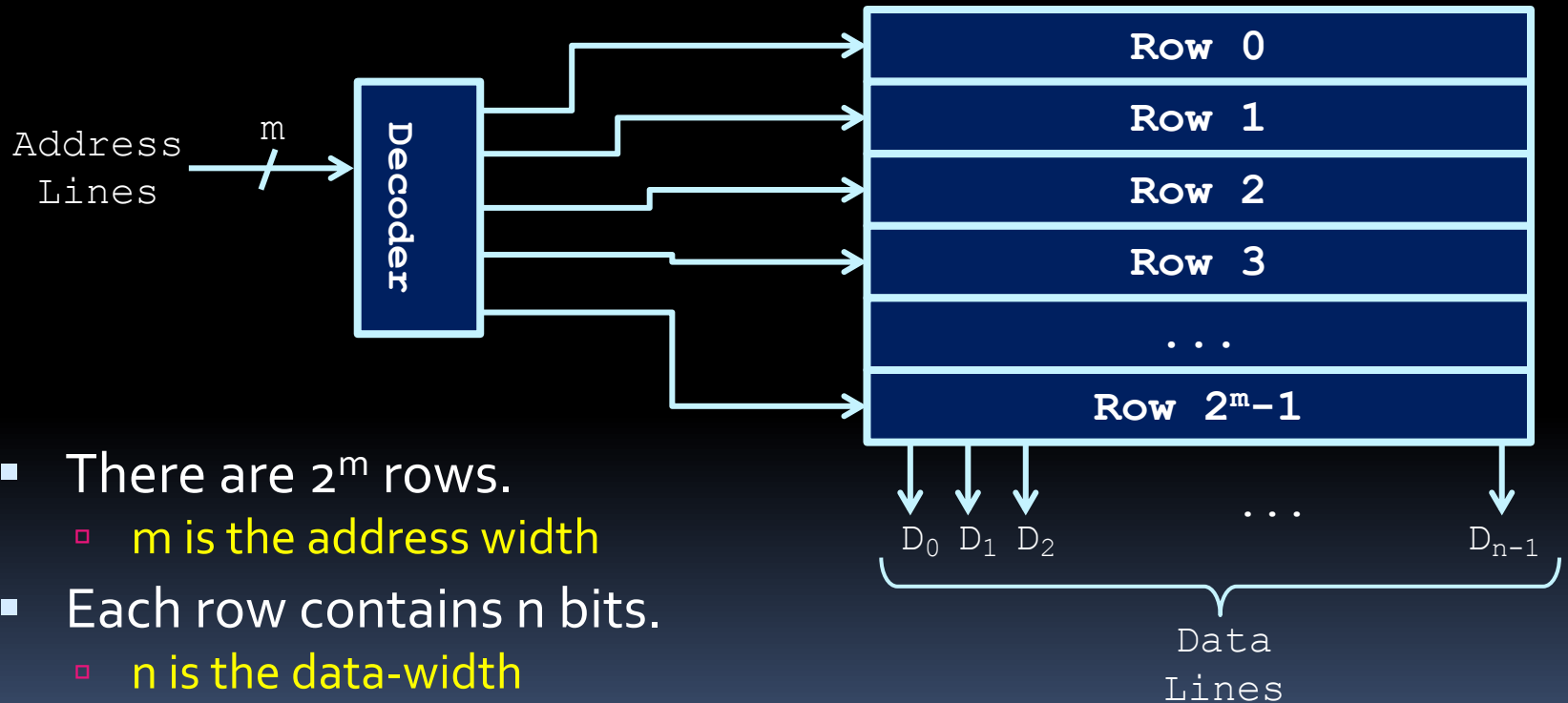
- Registers files are fast but too costly for storing lots of data.
- Instead store data in main memory.
- Main memory is **addressed in units of bytes** (8-bits)
- Every group of 4 bytes is one 32-bit **word**.

Address



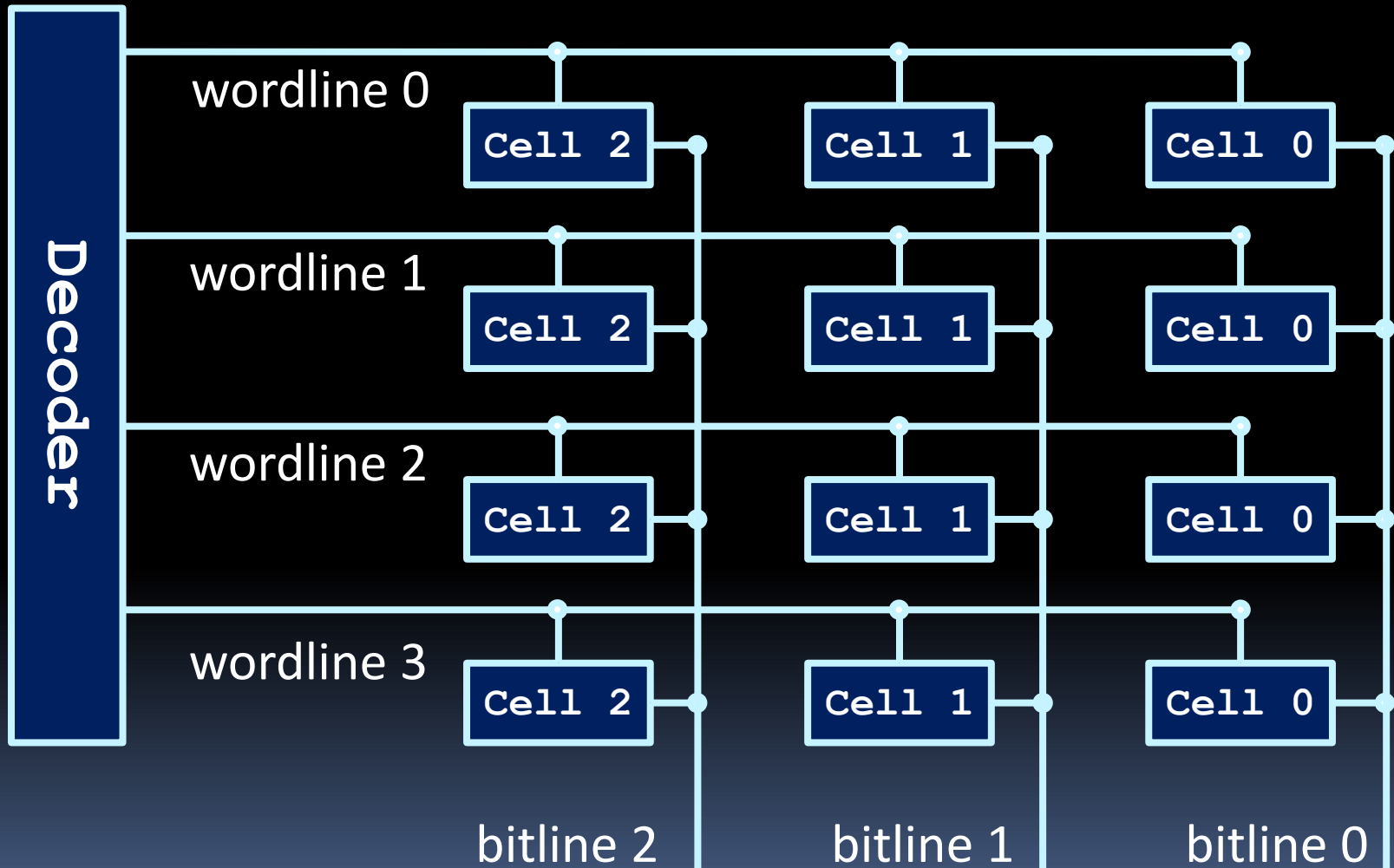
Electronic Memory

- Like register files, main memory is made up of a decoder and rows of memory units.



- There are 2^m rows.
 - m is the address width
- Each row contains n bits.
 - n is the data-width
- What's the size of this memory?
 - $2^m * n$ bits $\Rightarrow 2^m * n / 8$ Bytes

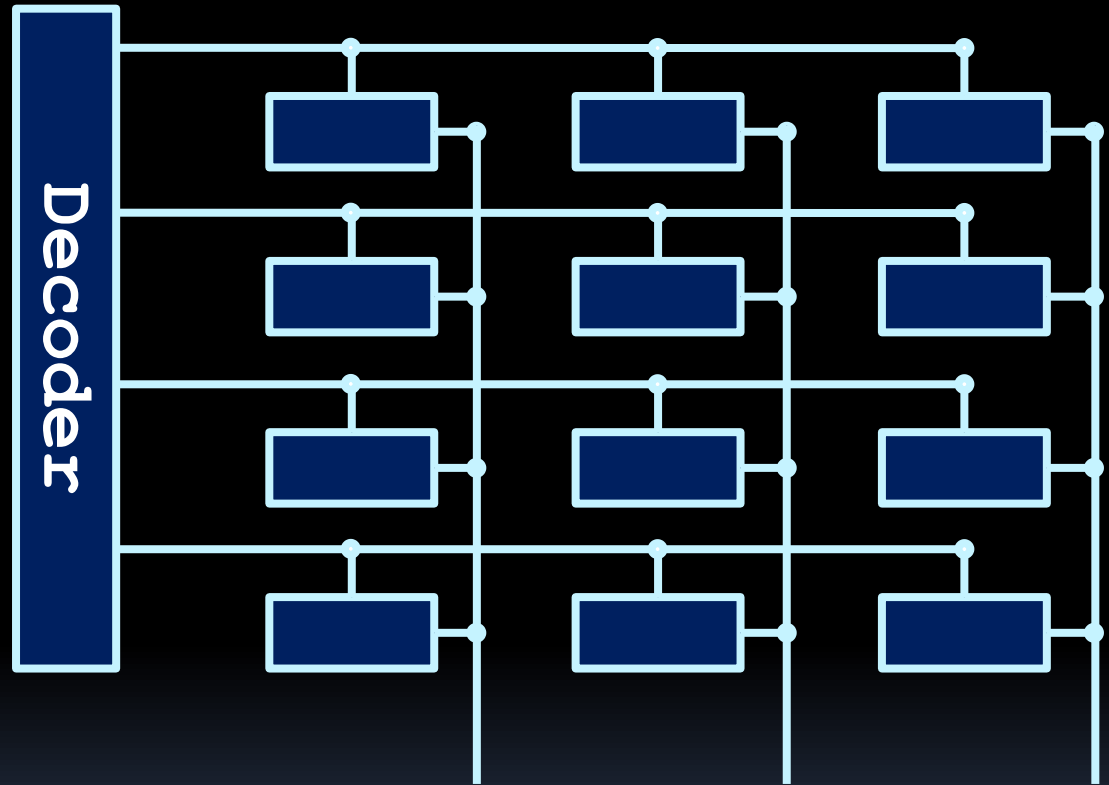
Memory Array



Memory Array – signals

Wordline:
which memory
row (word) to
read/write

Bitline:
read/write data

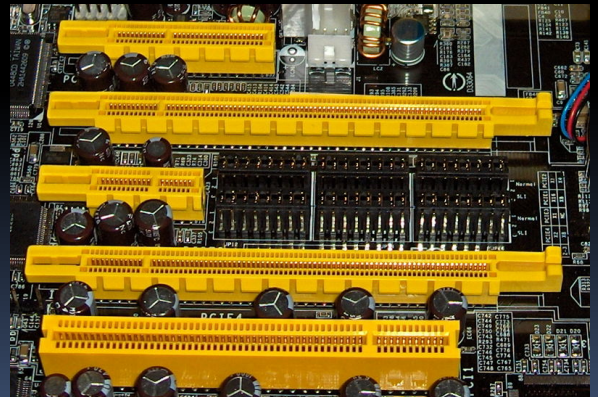


Also add **read/write** signal.

Can add column select line if needed.

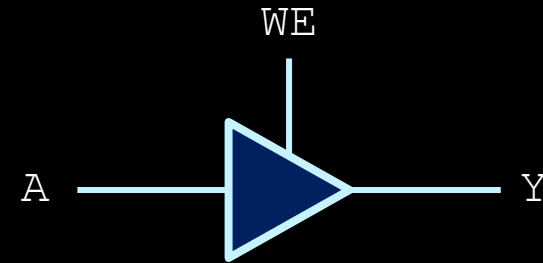
Data Bus

- Communication between components takes place through groups of shared wires called a **bus** (or **data bus**).
- Multiple components can read from a bus, but only one can write to a bus at a time.
 - Also called a **bus driver**.
- Each component has a **tristate buffer** that feeds into the bus. When not reading or writing, the tristate buffer drives high impedance onto the bus.



Controlling the flow

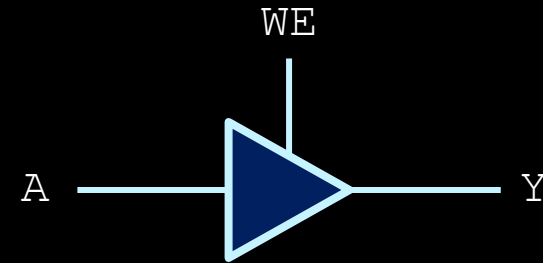
- Since some lines (buses) will now be used for both input and output, we introduce a (sort of) new gate called the **tri-state buffer**.
- When WE (write enable) signal is low, buffer output is a **high impedance** signal.
 - The output is neither connected to high voltage or to the ground.



WE	A	Y
0	x	z
1	0	0
1	1	1

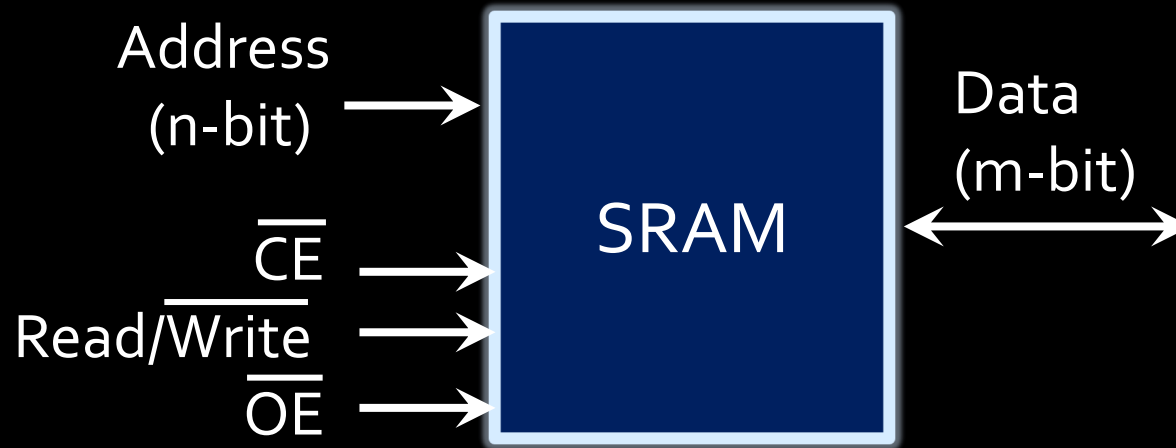
Controlling the flow

- $WE = 1$
 - A is connected to Y
- $WE = 0$
 - A is disconnected from Y
- Used to control data lines so that only one device can write onto the bus at any time (Multiple devices reading is usually fine)



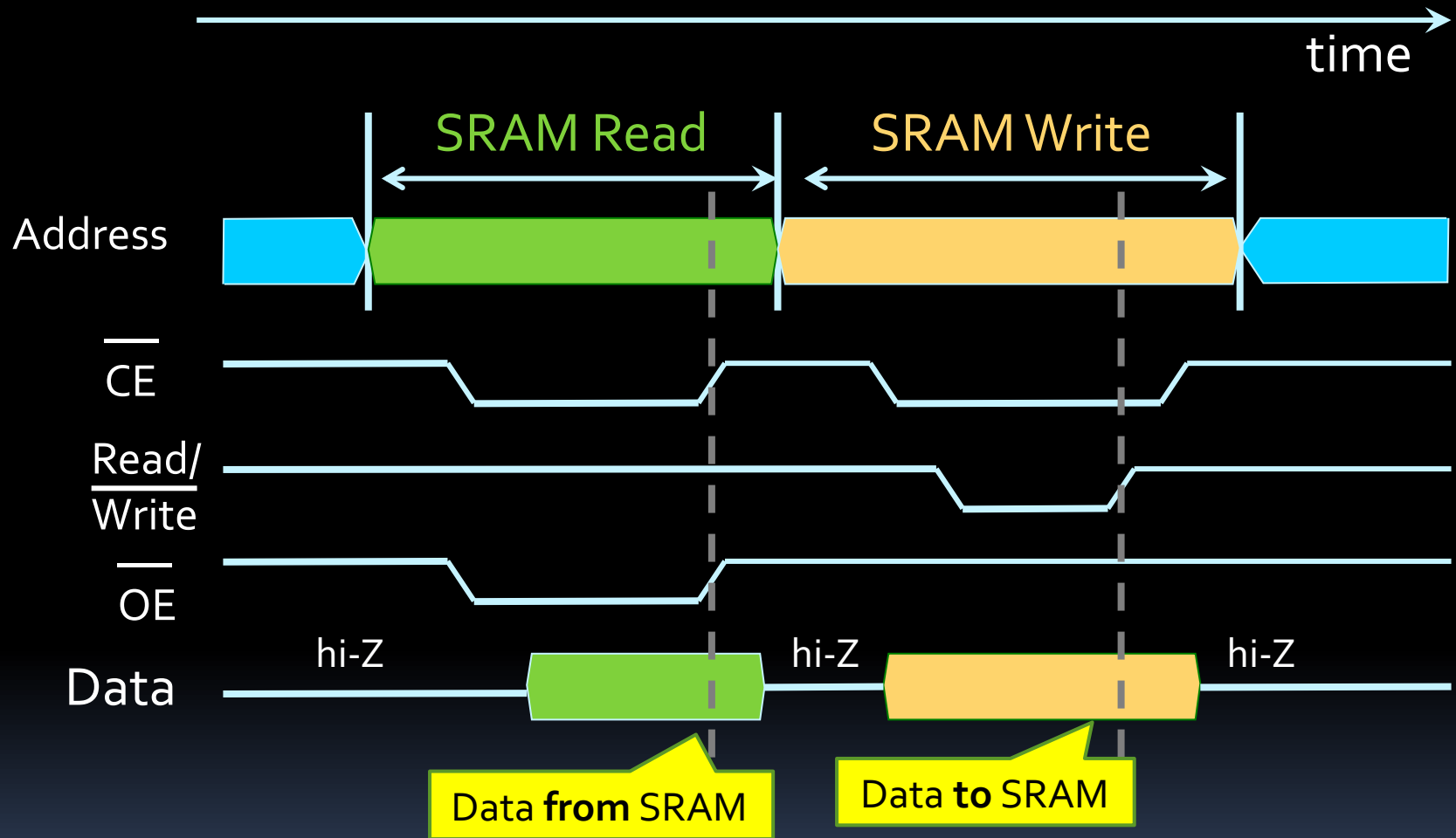
WE	A	Y
0	x	z
1	0	0
1	1	1

Example: Asynchronous SRAM Interface



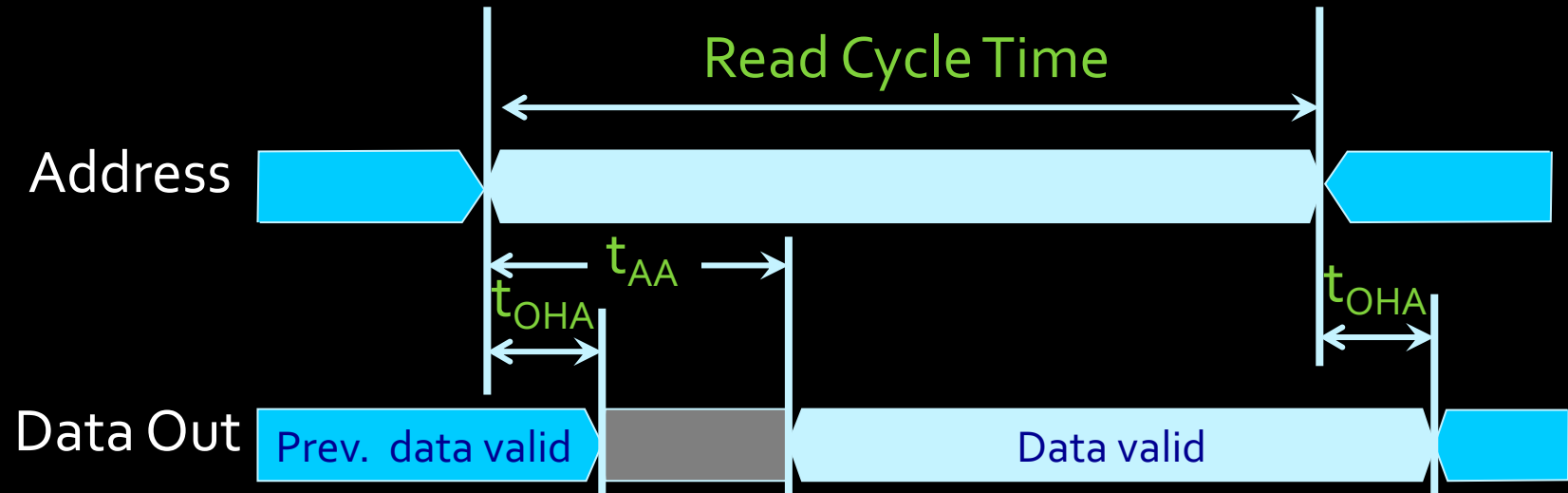
$\overline{\text{Chip Enable}}$ ($\overline{\text{CE}}$)	$\overline{\text{Read/Write}}$	$\overline{\text{Output Enable}}$ ($\overline{\text{OE}}$)	Access Type
0	0	1	SRAM Write
0	1	0	SRAM Read
1	X	X	SRAM not enabled

Asynchronous SRAM - Timing waveforms



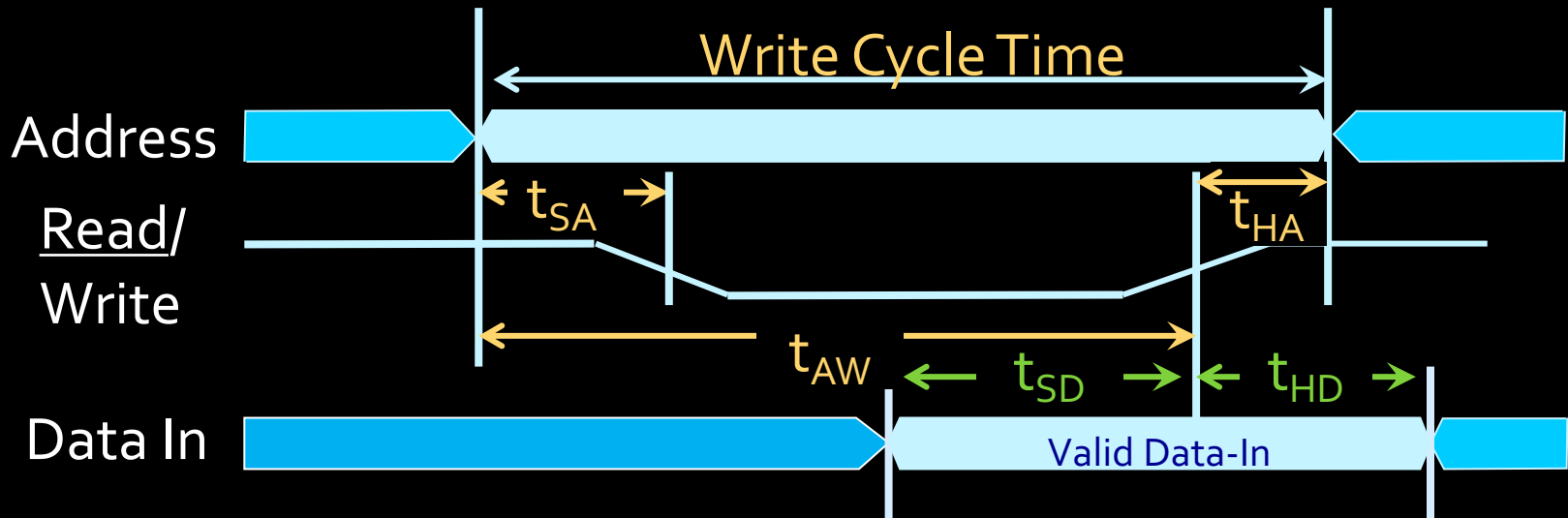
- Each memory read and write is done in stages.
- Each stage takes a certain amount of time.

Reading From Memory – Timing Constraints



- t_{AA} = **Address Access time**
 - Time needed for address to be stable before reading data values (~10ns).
- t_{OHA} = **Output Hold time**
 - Time output data is held after change of address (~2ns).

Writing To Memory – Timing Constraints



- t_{SA} = **Addr. Setup Time** (~0ns)
 - Time for address to be stable before enabling write signal.
- t_{AW} = **Address Setup Time to Write End** (~8ns)
- t_{SD} = **Data Setup to Write End** (~6ns)
 - Time for data-in value to be set-up at destination.
- t_{HD} = **Data Hold from Write End** (~0ns)
 - Time data-in value should stay unchanged after write signal changes.

Memory vs registers

- Memory houses most of the data values being used by a program.
- Registers are for local / temporary data stores, meant to be used to execute an instruction.
 - Registers are can host memory between instructions (like scrap paper for a calculation).
 - Some have special purpose or used to control execution, like the stack pointer register

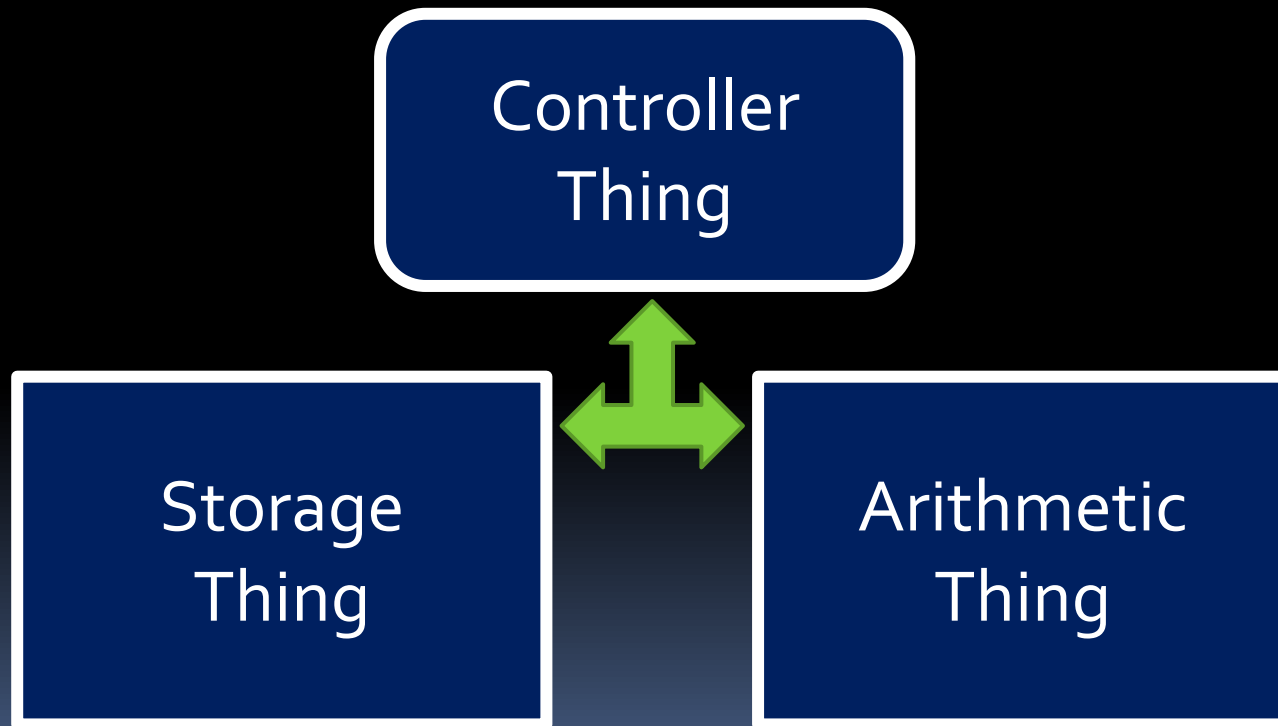
Memory vs registers

- In terms of access speed
 - Register = The plate in front of you
 - Cache = The fridge in the kitchen
 - Memory (RAM) = The corner grocery store
 - Hard Disk = The farm in the prairies
 - Network = The farm in another country

Load-Store Architecture

- The MIPS processor architecture we are building is a **load-store architecture**.
 - We load data from main memory to registers
 - Process them using ALU
 - Store back in main memory
- We do either ALU or memory, not both.
- This simplifies design of datapath and instruction set.

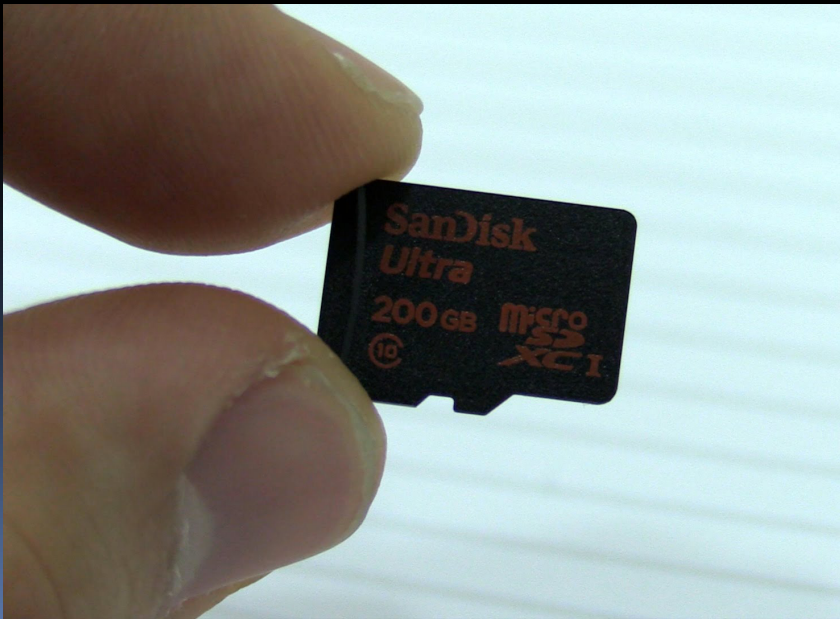
- Now we know what the Arithmetic and Storage Things do



Break

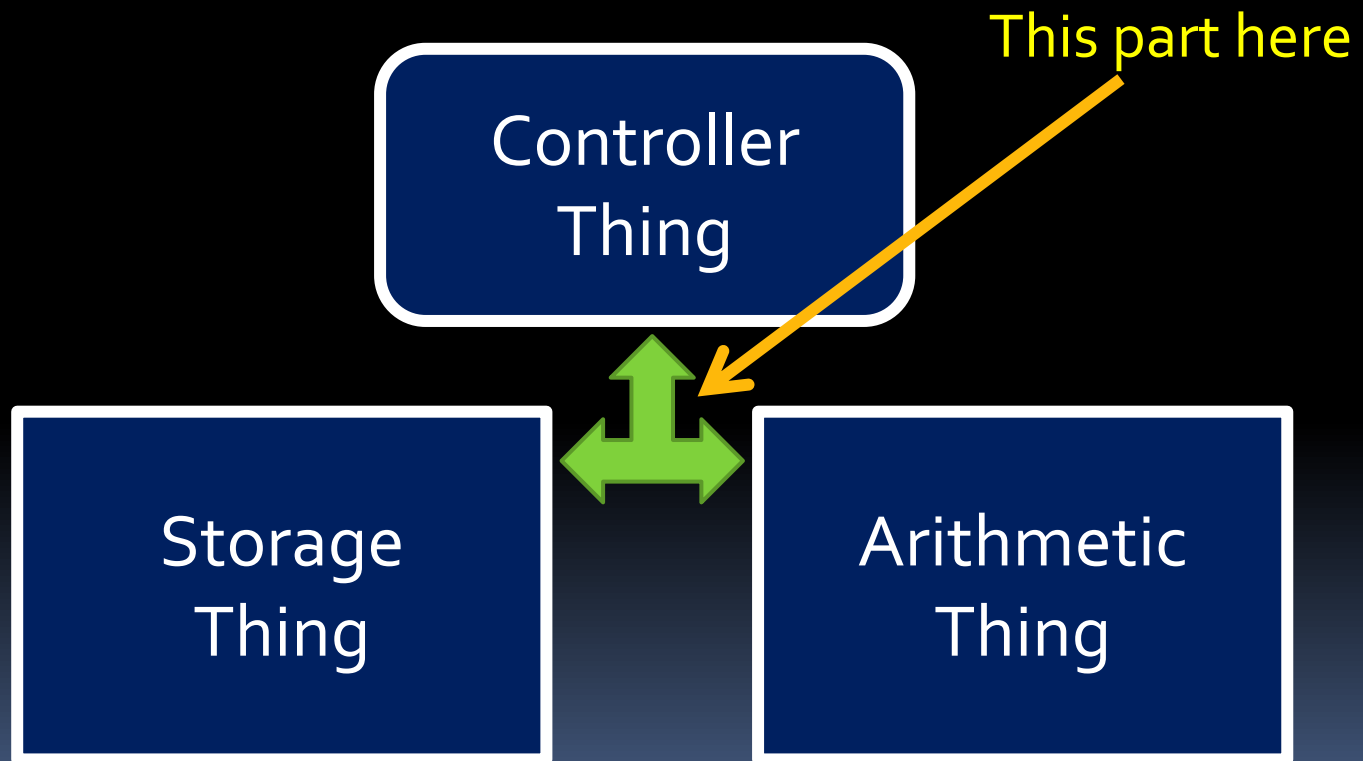


- 1981 :
- 2GB
 - 3MB/s
 - \$140,000

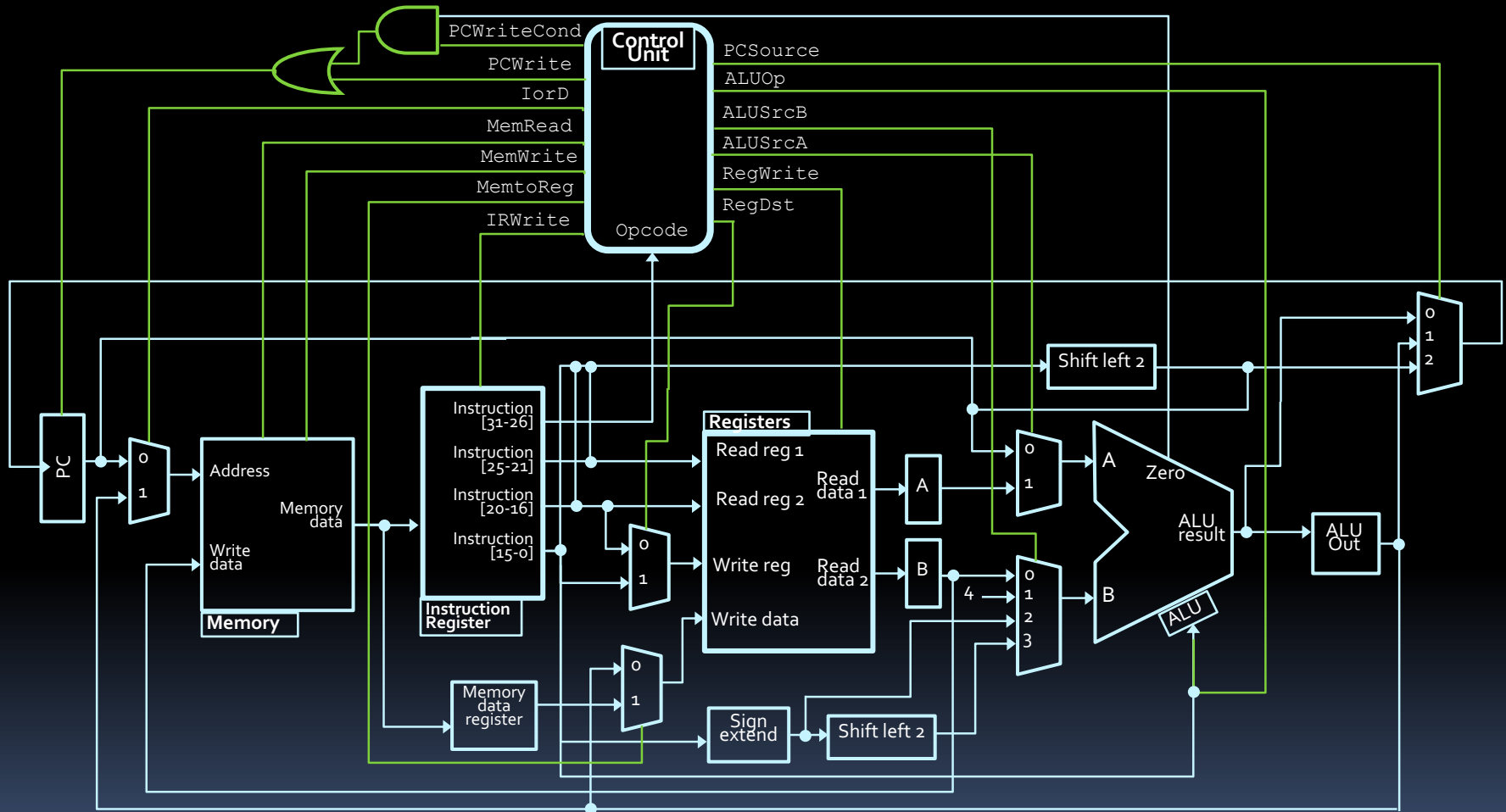


- 2019:
- 200GB
 - 50MB/s
 - \$25

- Before we get to the controller
- Need to talk about the data path



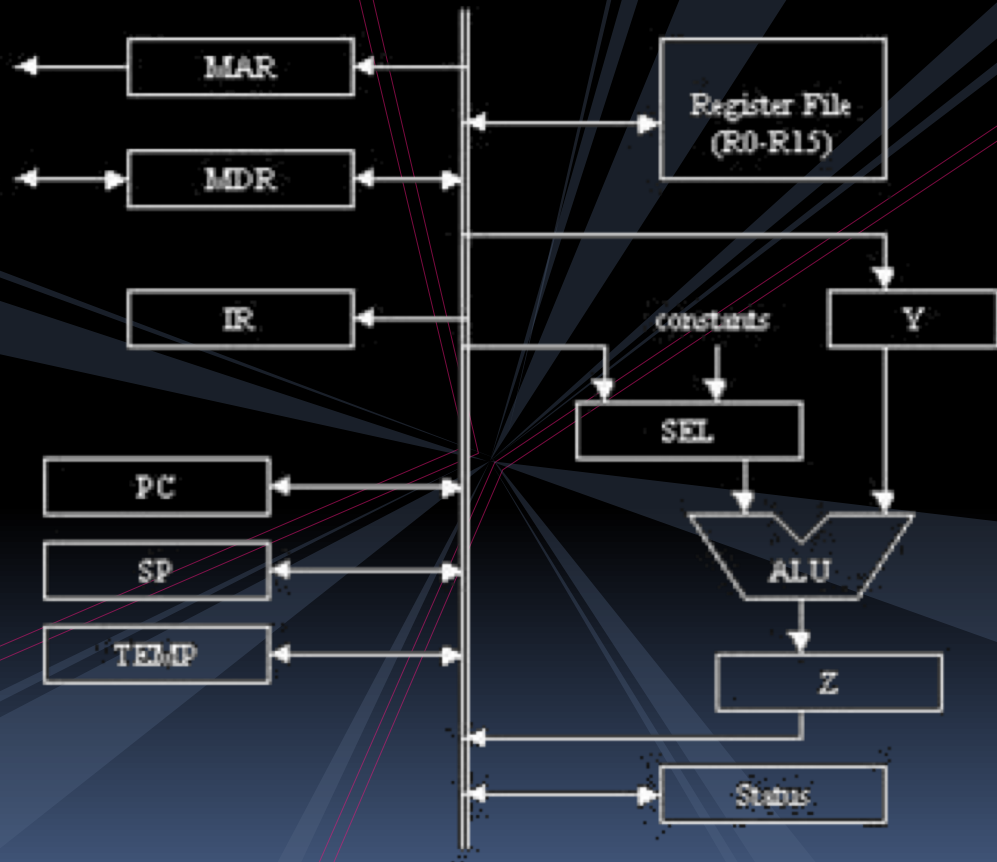
Processor Datapath Diagram



Datapath vs. Control

- **Datapath:** where all data computations take place.
 - Often a diagram version of real wired connections.
- **Control unit:** orchestrates the actions that take place in the datapath.
 - The control unit is a big finite-state machine that instructs the datapath to perform all appropriate actions.

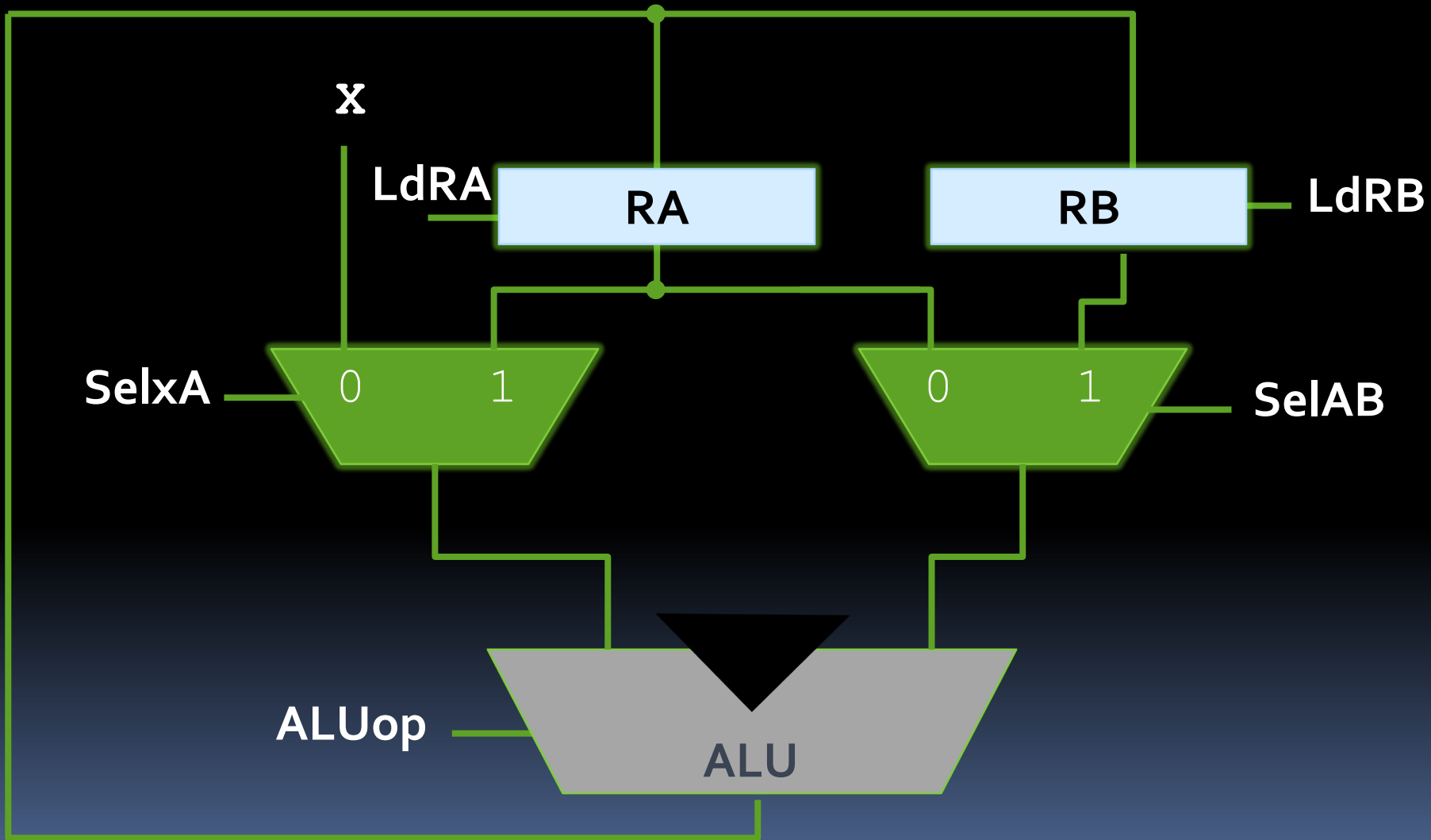
Datapath example



Example: Calculate $x^2 + 2x$

- Assume that you have access to a value from an external source. How would you compute $x^2 + 2x$ with components you've seen so far?
- Components needed:
 - **ALU** (to add, subtract and multiply values)
 - **Multiplexers** (to determine what the inputs should be to the ALU)
 - **Registers** (to hold values used in the calculation)

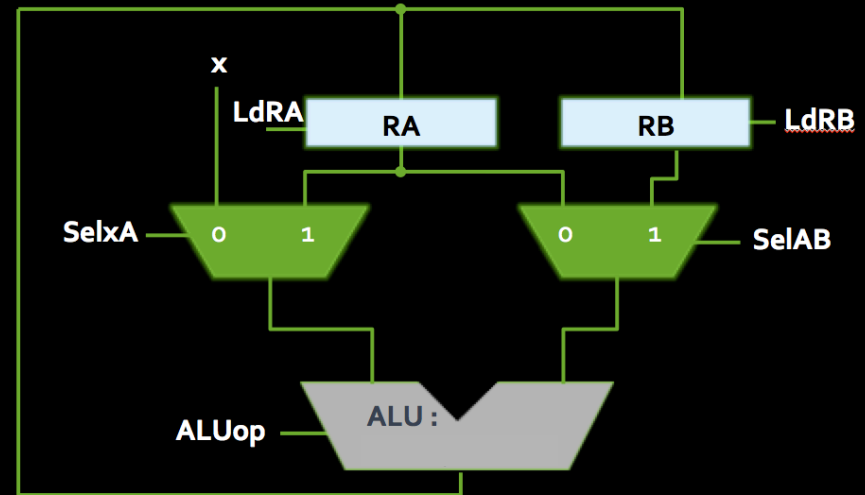
Example schematic



Making the calculation

Steps for $x^2 + 2x$:

- Load X into RA & RB
 - Store result in RA
 - Multiply RA & RB
 - Store result in RA
 - Add X to RA
 - Store result in RA
 - Add X to RA again
 - ALU output is $x^2 + 2x$.
-
- How do we make this happen?



Making the calculation

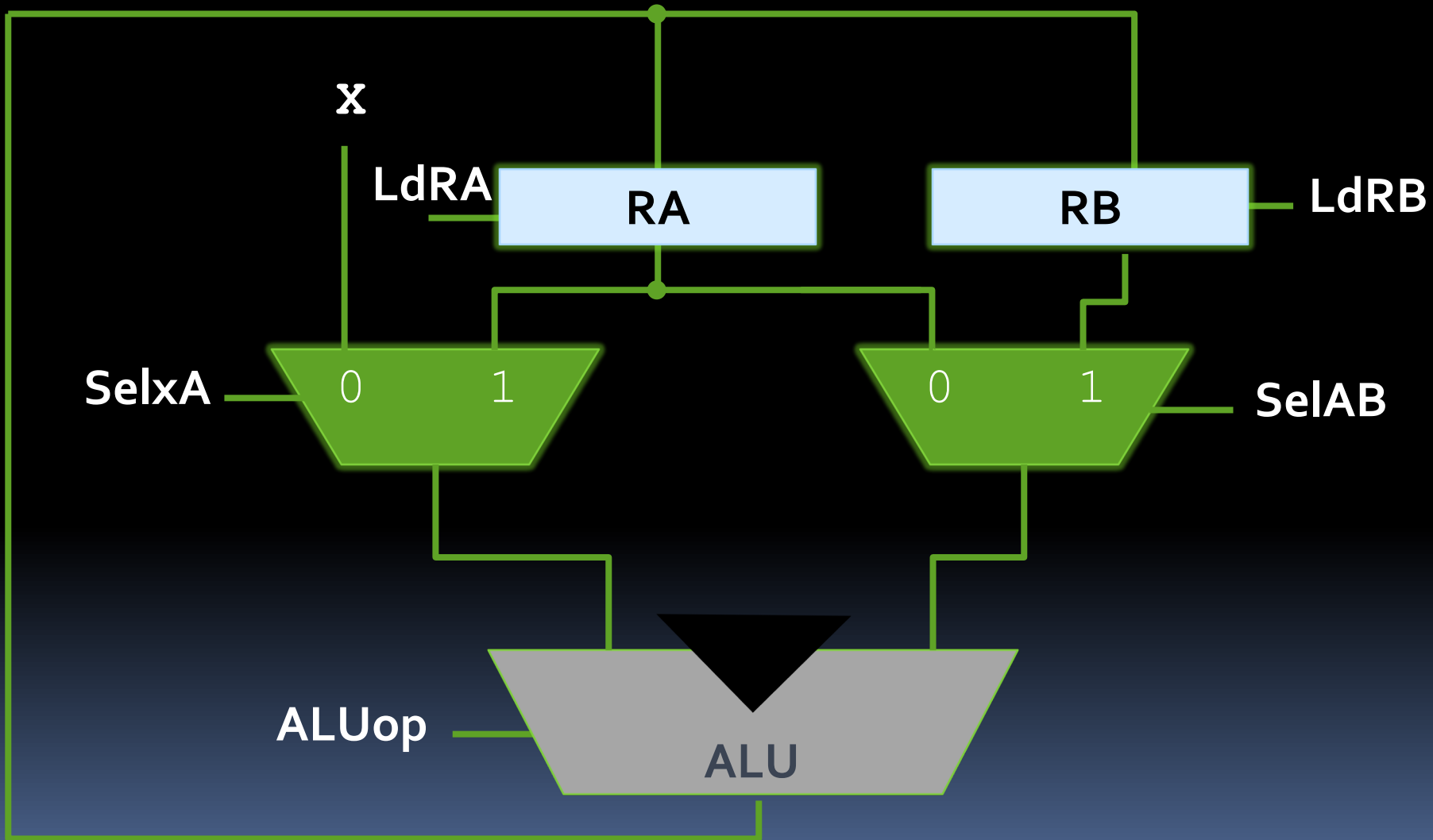
High-level Steps

- Load X into RA & RB
- Multiply RA & RB
 - Store result in RA
- Add X to RA
 - Store result in RA
- Add X to RA again
 - ALU output is $x^2 + 2x$.
- **Who sends these signals?**

Control Signals

- SelxA = 0, ALUop = Pass, LdRA = 1, LdRB = 1
- SelxA = 1, SelAB = 1, ALUop = Multiply, LdRA = 1
- SelxA = 0, SelAB = 0, ALUop = Add, LdRA = 1
- SelxA = 0, SelAB = 0, ALUop = Add

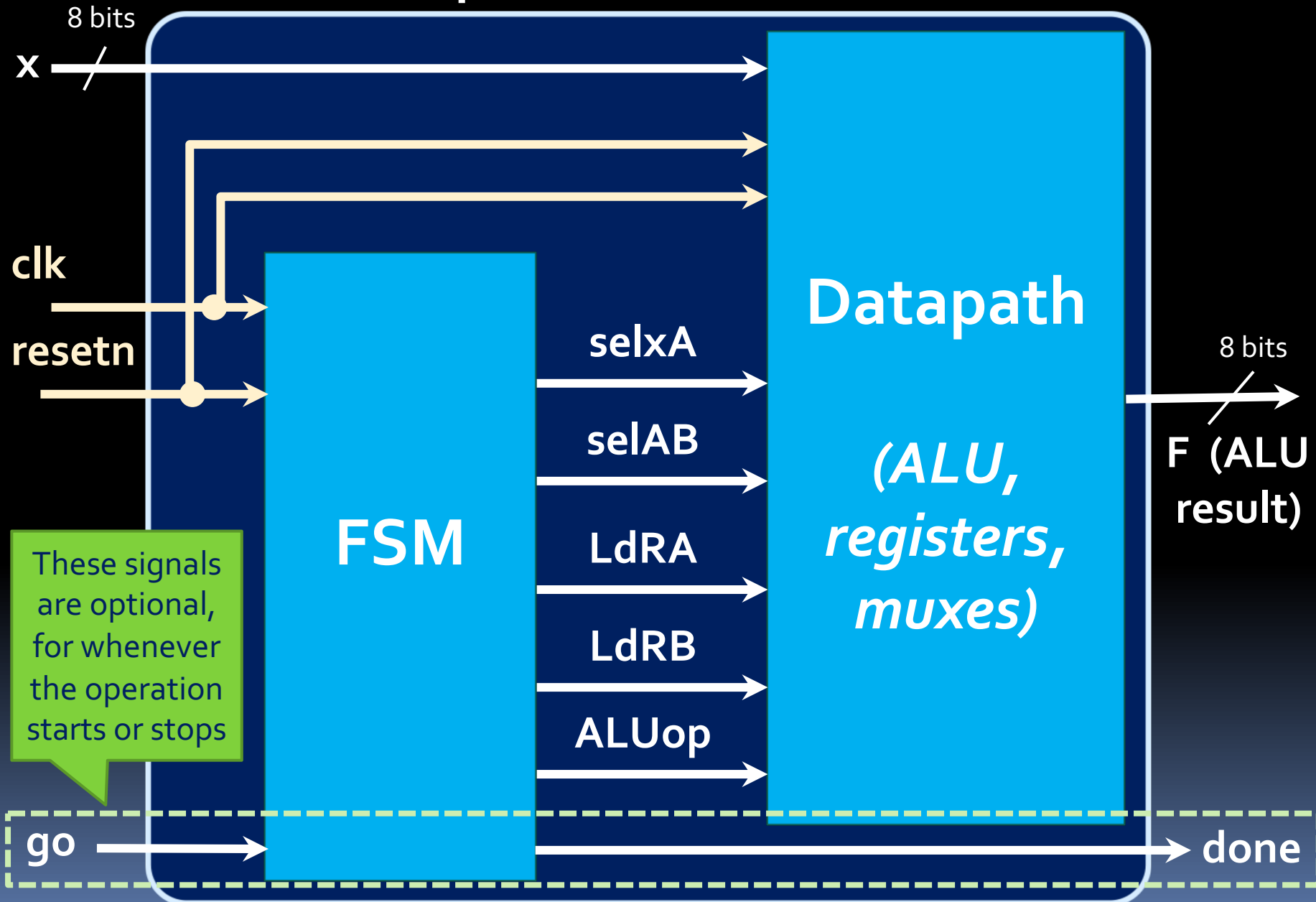
Example schematic



Control Unit

- Basically, a giant Finite State Machine
 - Synchronized to system-wide signals (**clock, resetn**)
- Outputs the **datapath control signals**
 - **SelxA, SelAB** => control mux outputs (ALU inputs)
 - **ALUop** => controls ALU operation
 - **LdRA, LdRB** => controls loading for registers RA, RB
- Some architectures also output a **done** signal, when the computation is complete
 - Yet another output; not shown in our datapaths

Datapath + Control



||| The “Control Thing”

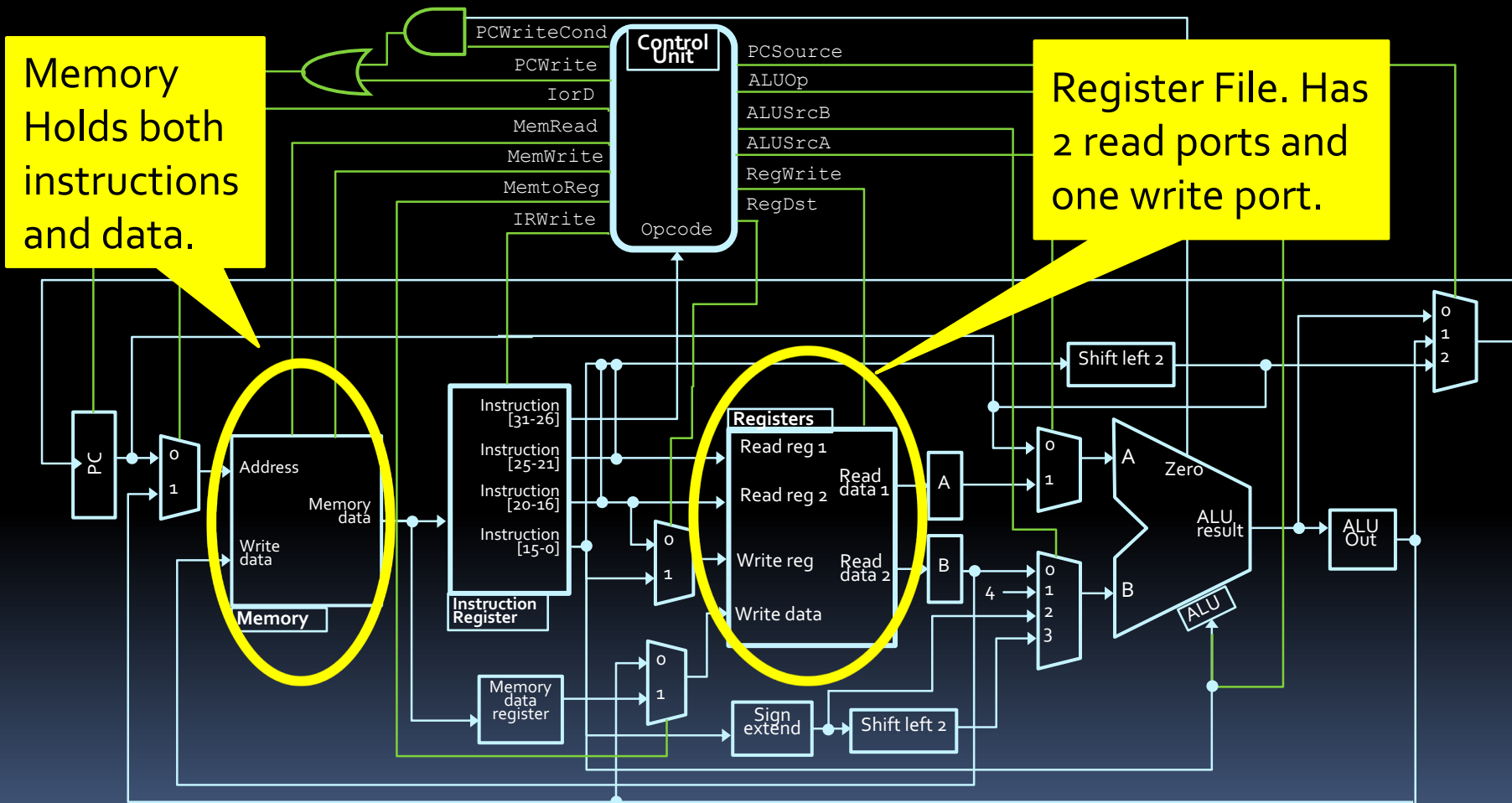
aka: the Control Unit



The Control Unit

- Control unit determines for data path:
 - Where the data is coming from (**the source**),
 - Where it's going to (**the destination**), and
 - How the data is being processed (**the operation**).
- How does the control unit know what operation to perform?
 - It gets information from an **instruction**.
 - This instruction also passes other information about the operation to the rest of the processor.
 - The control unit is responsible for loading the next instruction to run, after completing the current one.

Data sources and destinations



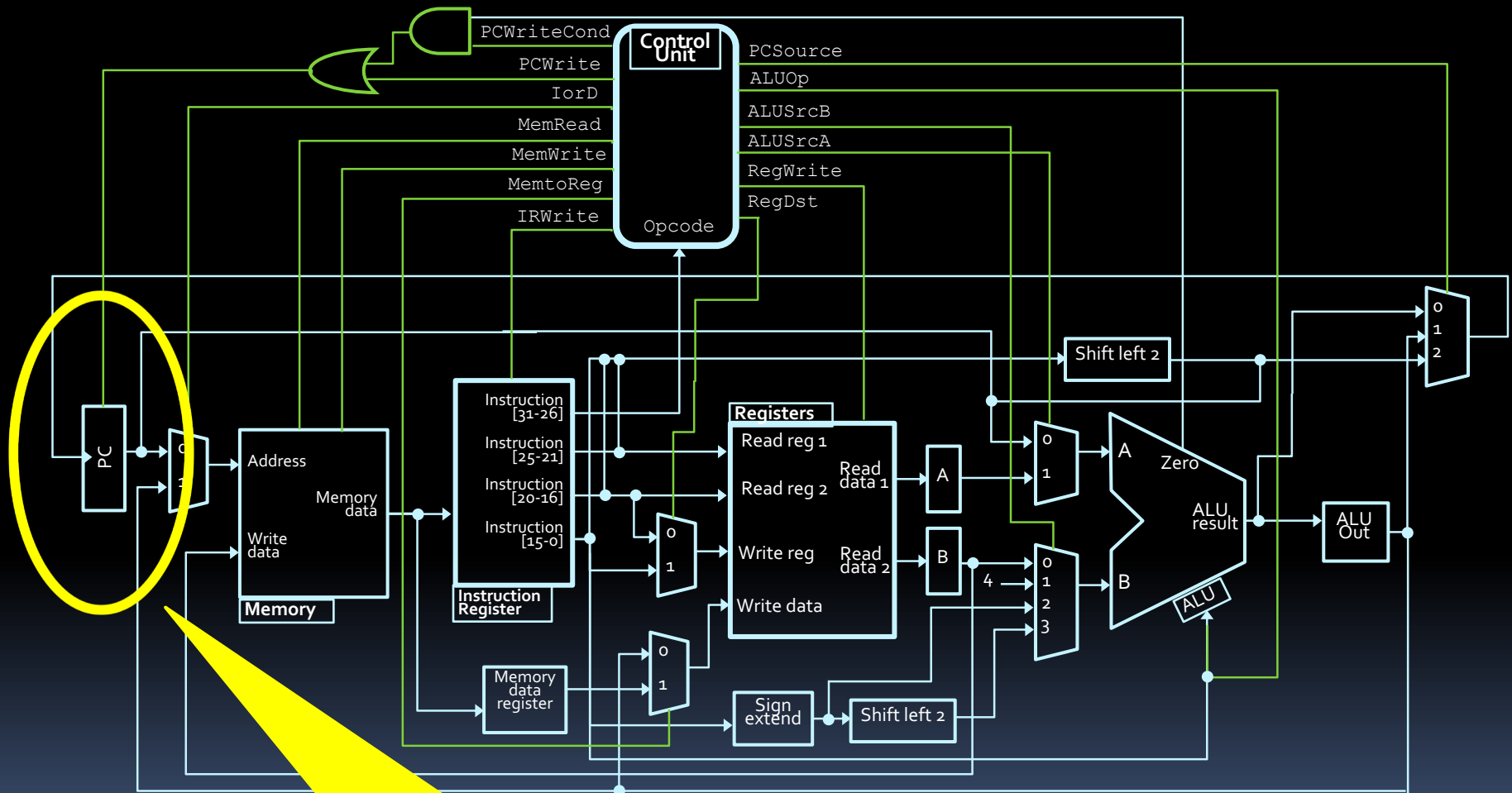
Executing a program

- What actually happens when you run an executable program on your computer? (e.g., Quartus.exe, ls, FaceTubeSnapFlix App)
 - OS loads a series of instructions into memory
 - Location of first instruction is provided to CPU
 - CPU executes instructions one at a time

Instructions

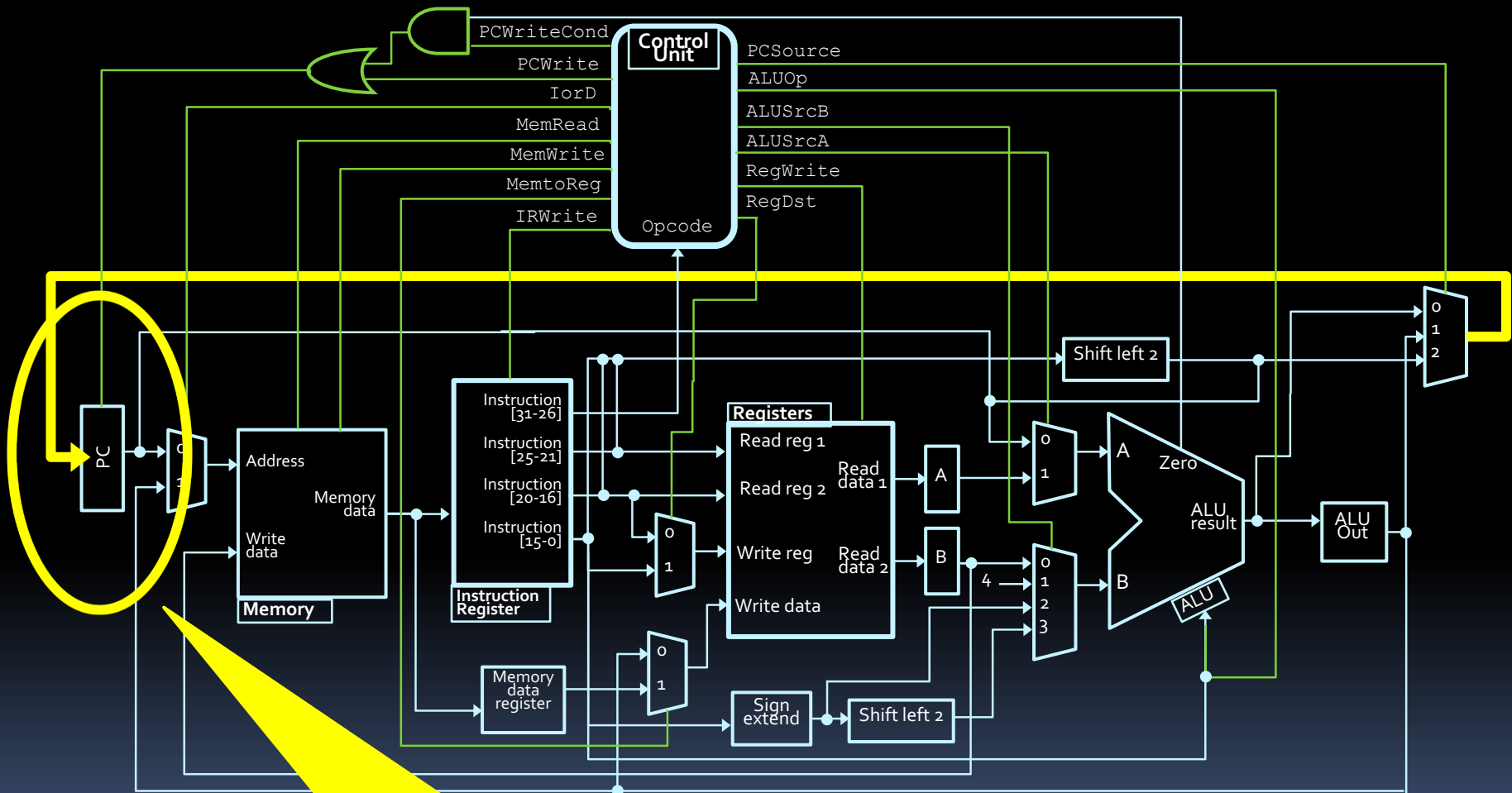
- What is an instruction?
 - 32 bit binary string in our MIPS processor
 - Length depends on architecture.
 - Tells the processor (the control thing) what to do
- How do we know which instruction to execute?
 - Special register: Program Counter (PC)
 - Incremented by 4 (32 bits = 4 bytes) after every instruction fetch
 - Can also be set by output of ALU to allow us to 'jump' to another part of the code

The Program Counter



Program counter holds address of instruction

The Program Counter



Program counter holds address of instruction

Instruction decoding

- Okay... Here's your instruction... GO

```
00000000 00000001 00111000 00100011
```



Instruction decoding

- The instructions themselves can be broken down into sections that contain all the information needed to execute the operation.
 - Also known as a **control word**.
- Example: unsigned subtraction

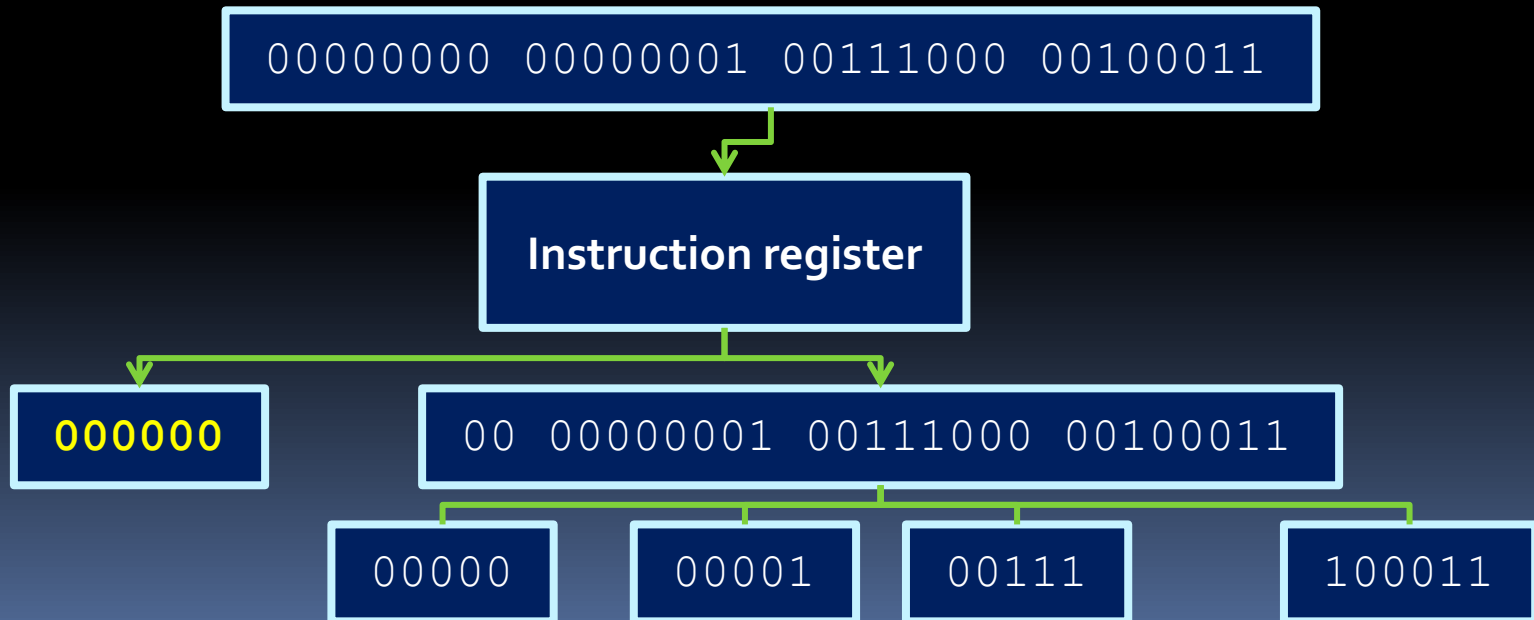
```
00000000 00000001 00111000 00100011
```

```
000000ss sssttttt dddd000000100011
```

```
Register 7 = Register 0 - Register 1
```

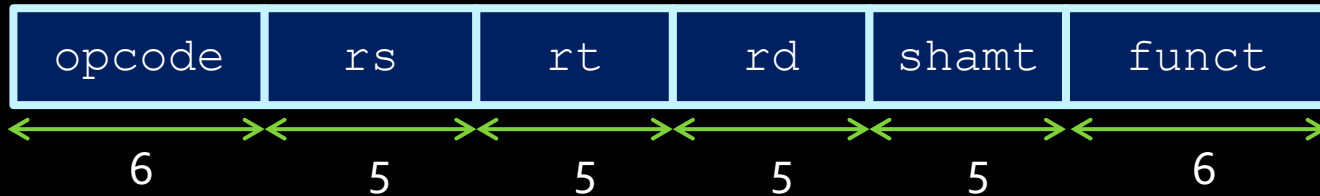
Instruction registers

- The **instruction register** takes in the 32-bit instruction fetched from memory, and reads the first 6 bits (known as the **opcode**) to determine what operation to perform.

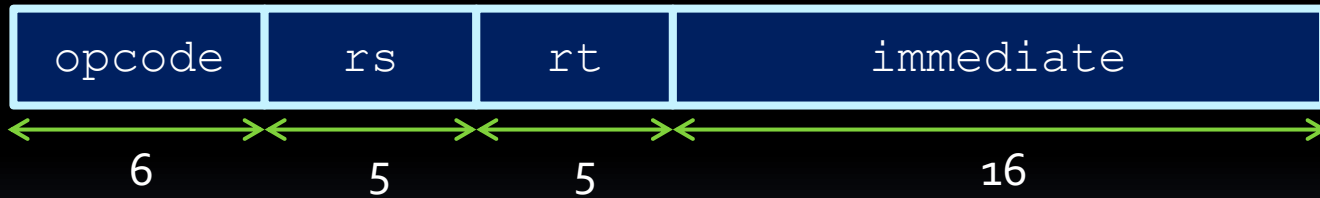


MIPS instruction types

- **R-type:**



- **I-type:**



- **J-type:**

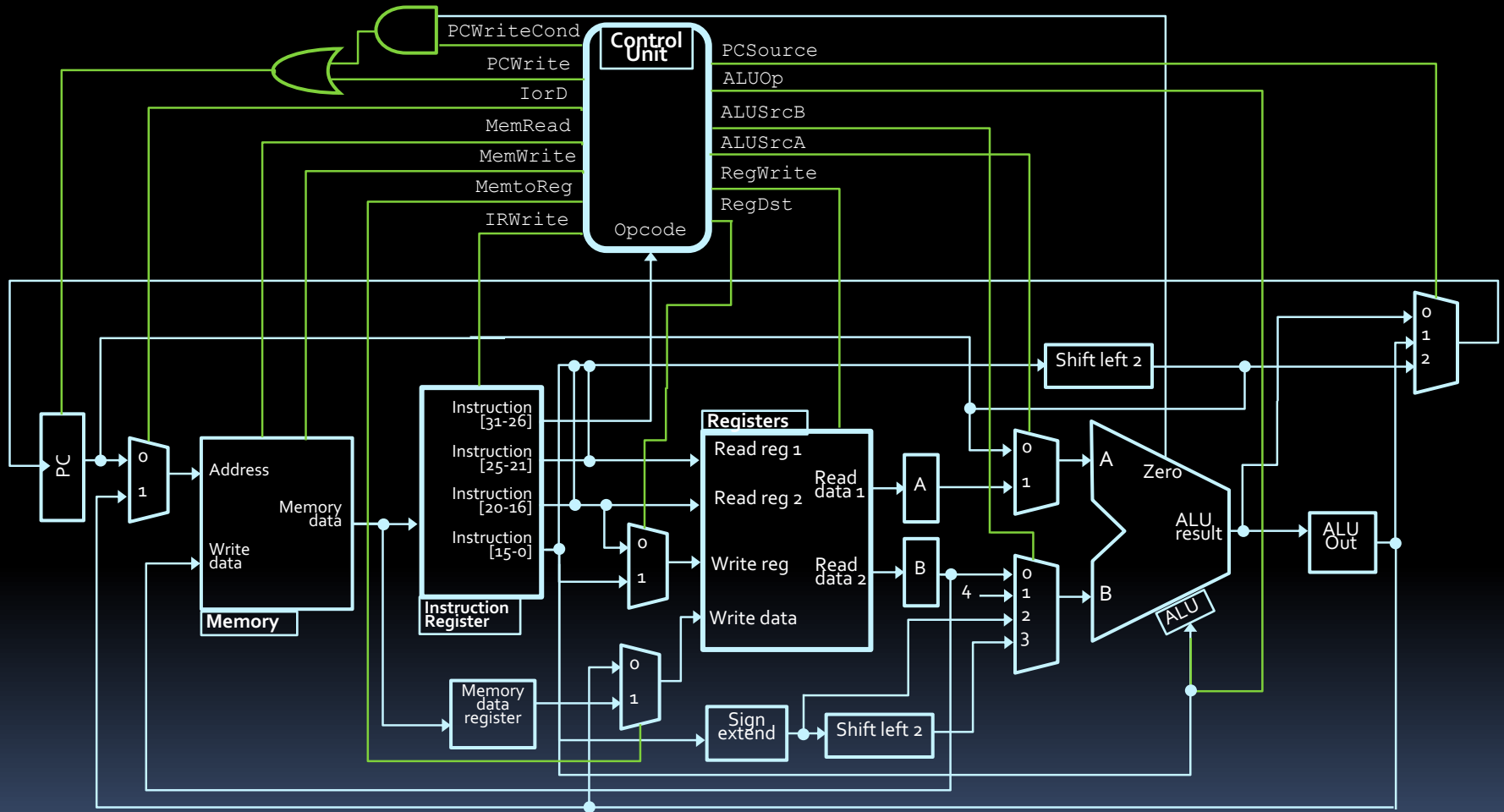


Opcodes

- The first six digits of the instruction (the opcode) will determine the instruction type.
 - For “R-type” instructions (marked in yellow) opcode is 000000, and last six digits denote the function.

<u>Instruction</u>	<u>Op/Func</u>	<u>Instruction</u>	<u>Op/Func</u>
add	100000	srav	000111
addu	100001	srl	000010
addi	001000	srlv	000110
addiu	001001	beq	000100
div	011010	bgtz	000111
divu	011011	blez	000110
mult	011000	bne	000101
multu	011001	j	000010
sub	100010	jal	000011
subu	100011	jalr	001001
and	100100	jr	001000
andi	001100	lb	100000
nor	100111	lbu	100100
or	100101	lh	100001
ori	001101	lhu	100101
xor	100110	lw	100011
xori	001110	sb	101000
sll	000000	sh	101001
sllv	000100	sw	101011
sra	000011	mflo	010010

The Processor Datapath

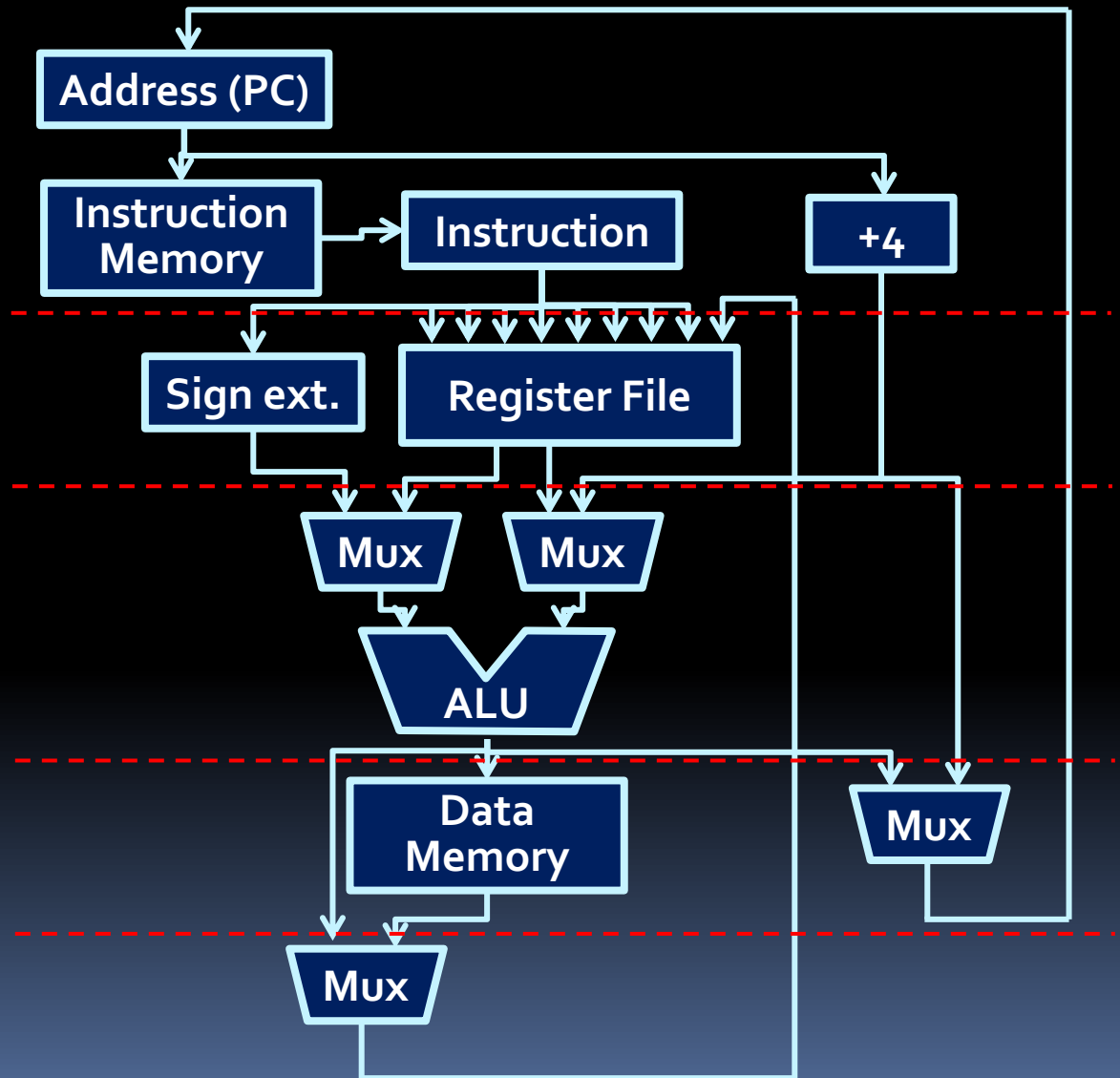


Simplified Datapath

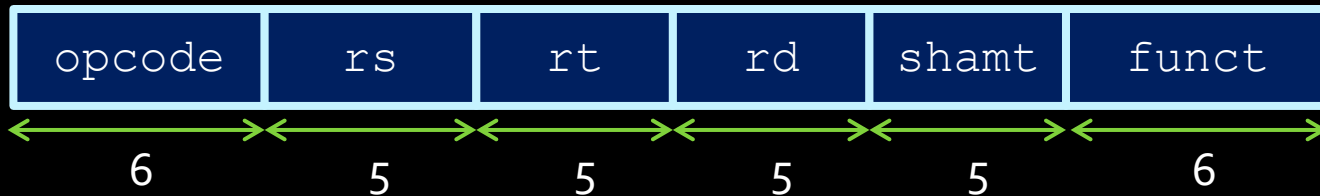
Note: this is just an abstraction 😊

- Most processor operations have stages as shown in the diagram:

1. Instruction fetch
2. Instruction decode & register fetch
3. Execute address/data calculation
4. Memory access
5. Write back.



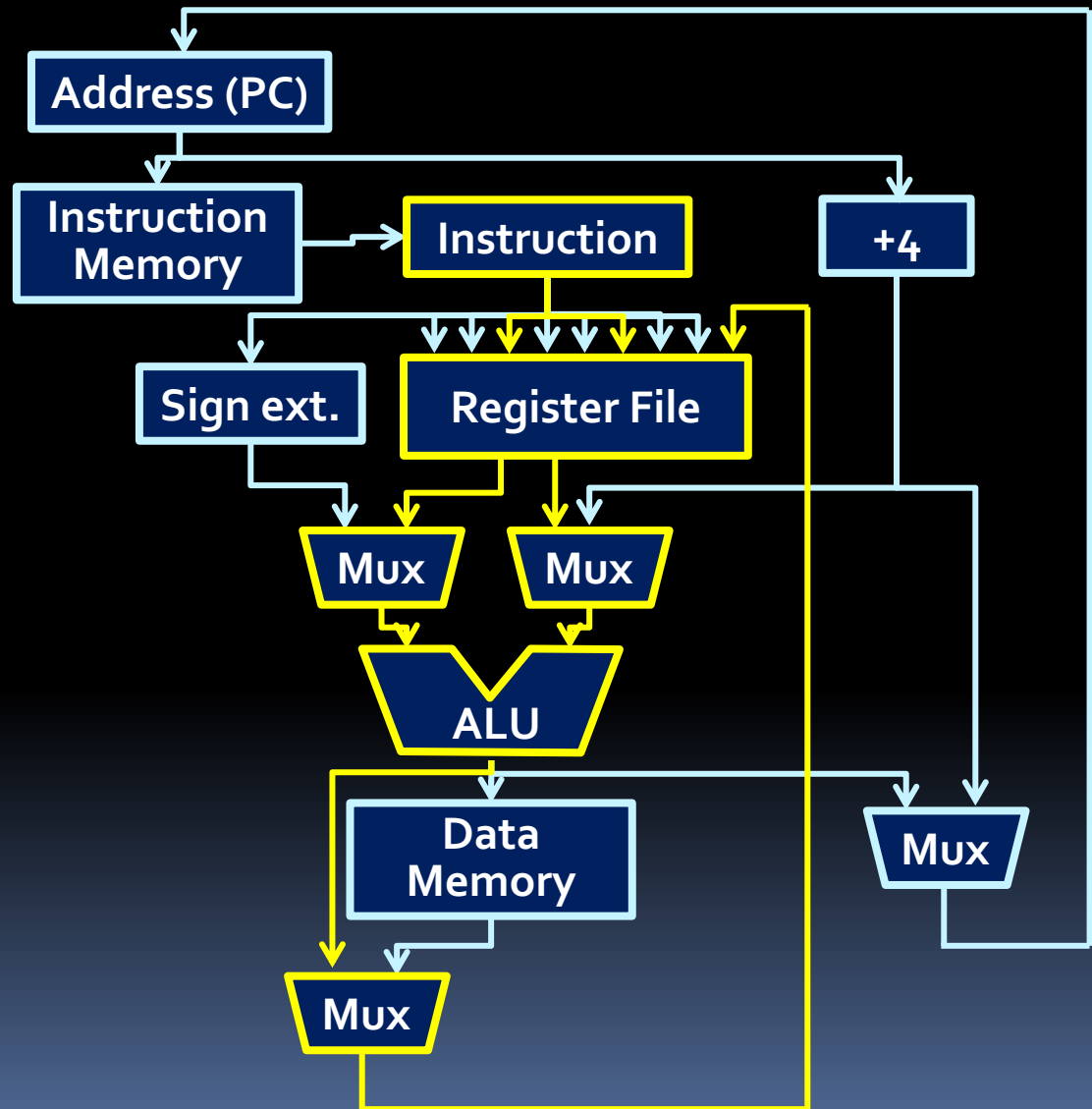
R-type instructions



- Short for “register-type” instructions.
 - Because they operate on the registers, naturally.
- These instructions have fields for specifying up to three registers and a shift amount.
 - Three registers: two source registers (*rs* & *rt*) and one destination register (*rd*).
 - A field is usually coded with all 0 bits when not being used.
- The opcode for all R-type instructions is 000000.
- The function field specifies the type of operation being performed (add, sub, and, etc).

R-type instruction datapath

- For the most part, the `funct` field tells the ALU what operation to perform.
- `rs` and `rt` are sent to the register file, to specify the ALU operands.
 - Register `$0` and `$1` are usually held in reserve.
- `rd` is also sent to the register file, to specify the location of the result.

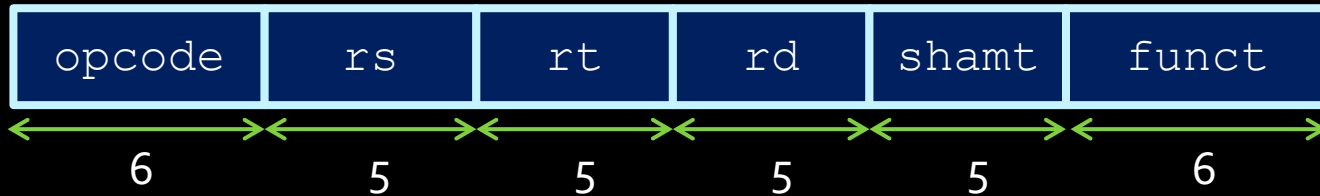


Example

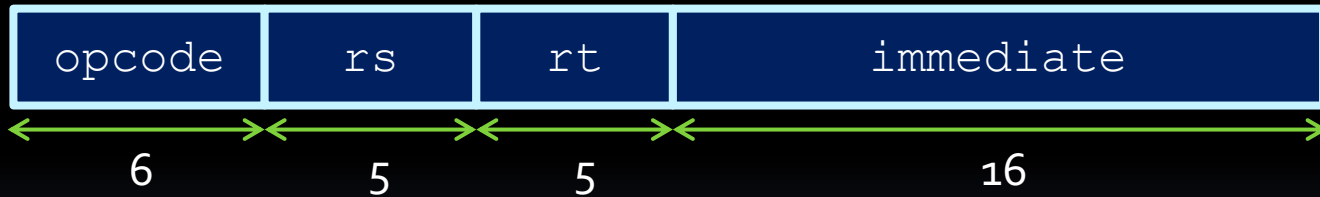
```
00000000 11010001 00101000 00100110
```

MIPS instruction types

- **R-type:**



- **I-type:**



- **J-type:**



<u>Instruction</u>	<u>Op/Func</u>	<u>Instruction</u>	<u>Op/Func</u>
add	100000	srav	000111
addu	100001	srl	000010
addi	001000	srlv	000110
addiu	001001	beq	000100
div	011010	bgtz	000111
divu	011011	blez	000110
mult	011000	bne	000101
multu	011001	j	000010
sub	100010	jal	000011
subu	100011	jalr	001001
and	100100	jr	001000
andi	001100	lb	100000
nor	100111	lbu	100100
or	100101	lh	100001
ori	001101	lhu	100101
xor	100110	lw	100011
xori	001110	sb	101000
sll	000000	sh	101001
sllv	000100	sw	101011
sra	000011	mflo	010010

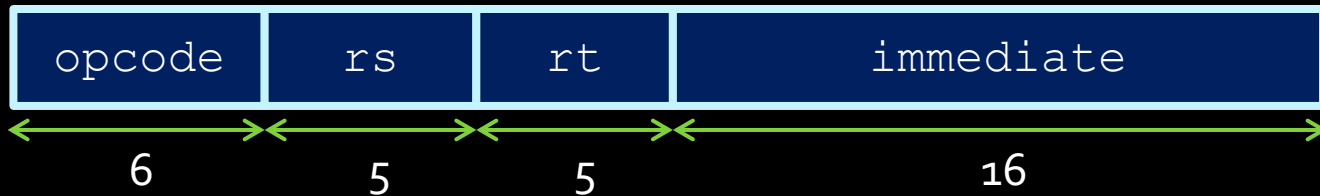
Example

```
00000000 11010001 00101000 00100110
```

- We look at opcode, it is 00000 → R-type
- Now look at **funct** → 100110 → XOR
- Now we look at **rs**, **rt**, and **rd** registers:
 - **rs = 6**, **rt = 17**, **rd = 5**

→ XOR the the value of registers 5 and 17 and store it in register 5 (**xor \$5, \$17, \$6**)

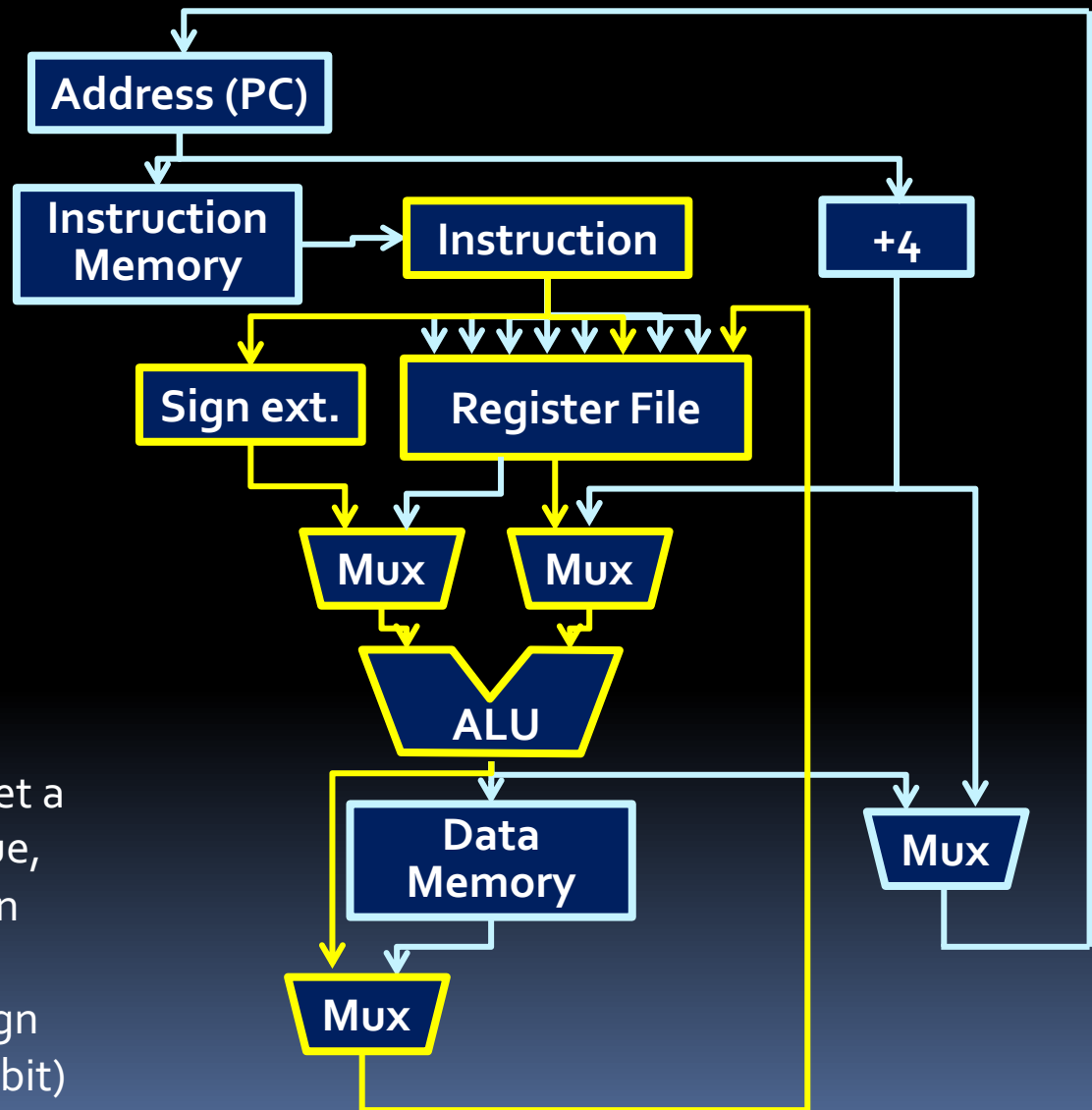
I-type instructions



- These instructions have a 16-bit **immediate** field.
- This field a constant value, which is used for:
 - an immediate operand,
 - a branch target offset (e.g., branch if equal)
 - a displacement for a memory operand. (e.g., load)
- For branch target offset operations, the immediate field contains the signed difference between the current address stored in the PC and the address of the target instruction.
 - This offset is stored with the two low order bits dropped. Since we can only jump by 4 bytes at a time (word alignment), we don't bother writing the lowest 2 bits, leaving more space for useful bits.

I-type instruction datapath

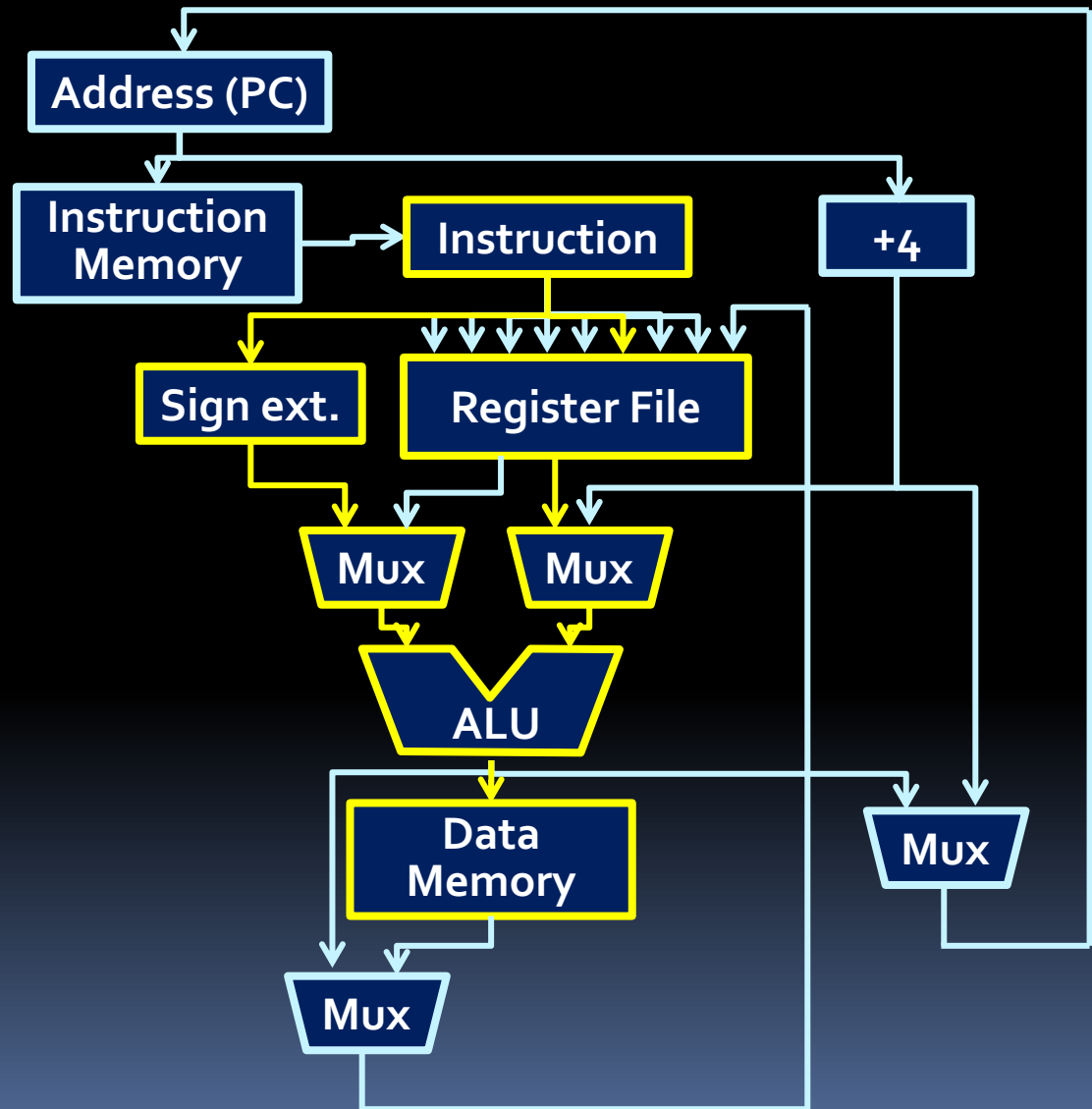
- Example #1: Immediate arithmetic operations, with result stored in registers.



- Sign Extension: We get a 16 bit immediate value, but need 32 bits for an ALU operand. So fill upper 16 bits with "sign bit" (most significant bit)

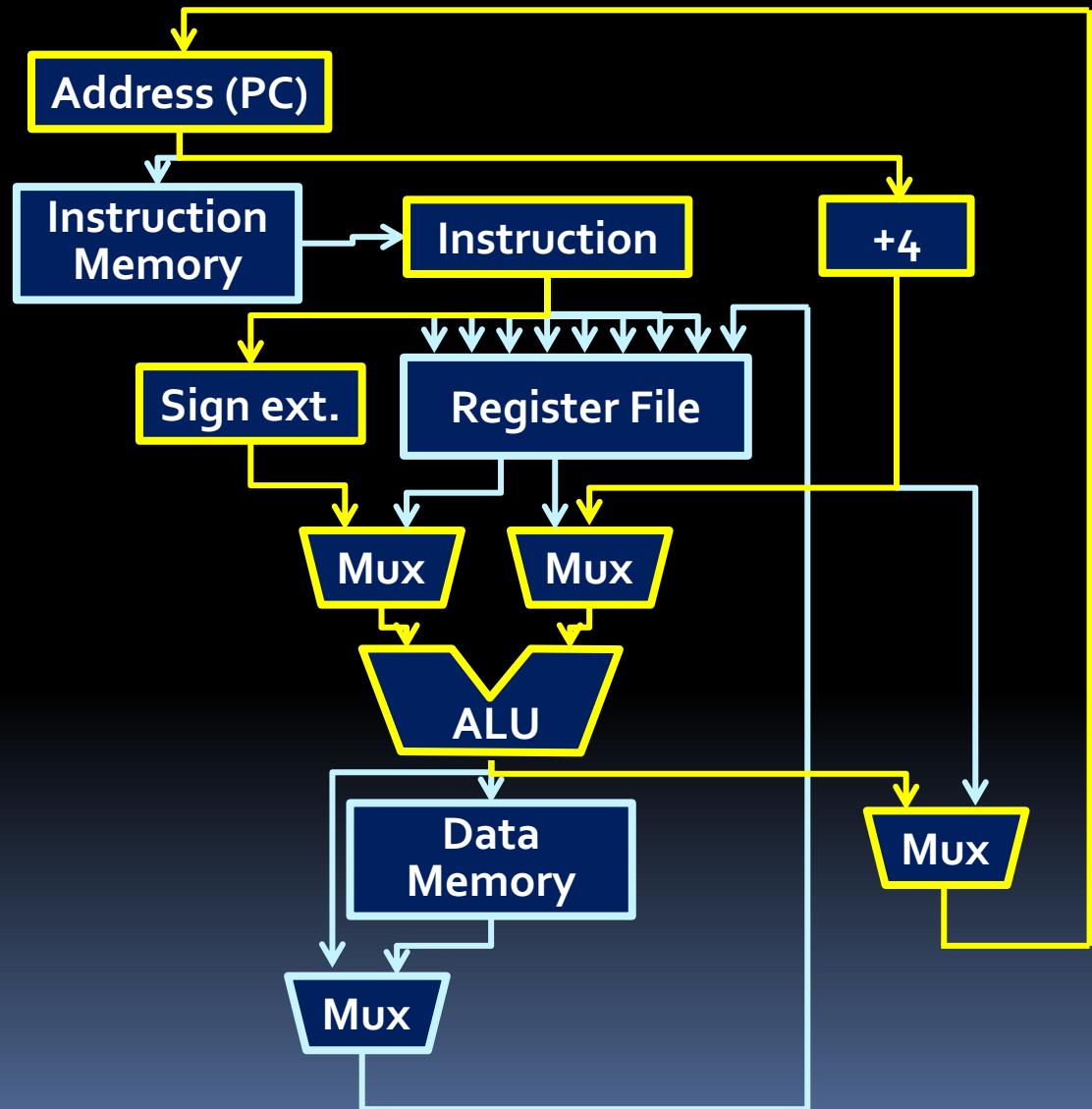
I-type instruction datapath

- Example #2: Immediate arithmetic operations, with result stored in memory.



I-type instruction datapath

- Example #3: Branch instructions.
 - Output is written to PC, which looks to that location for the next instruction.



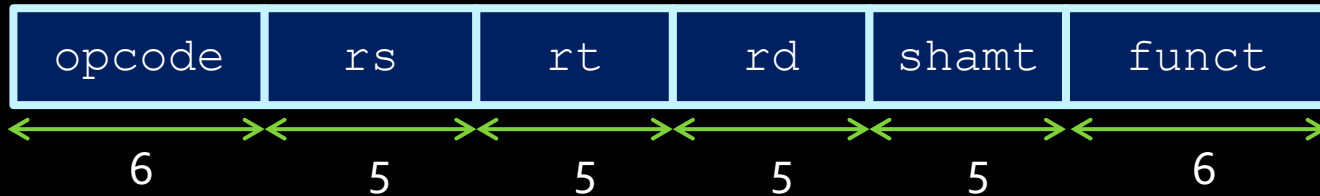
Example

```
00100000 11010001 00000000 00100110
```

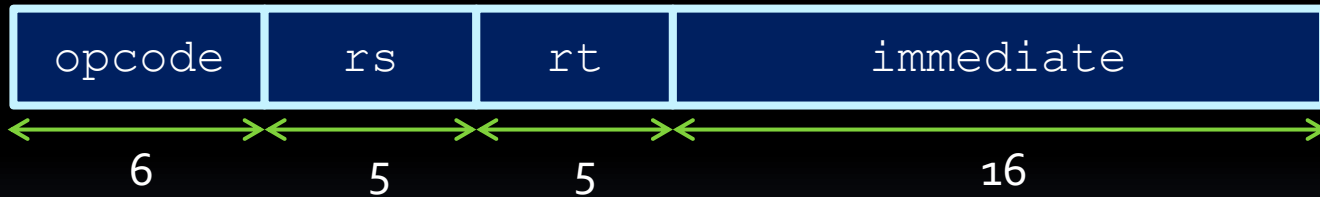
<u>Instruction</u>	<u>Op/Func</u>	<u>Instruction</u>	<u>Op/Func</u>
add	100000	srav	000111
addu	100001	srl	000010
addi	001000	srlv	000110
addiu	001001	beq	000100
div	011010	bgtz	000111
divu	011011	blez	000110
mult	011000	bne	000101
multu	011001	j	000010
sub	100010	jal	000011
subu	100011	jalr	001001
and	100100	jr	001000
andi	001100	lb	100000
nor	100111	lbu	100100
or	100101	lh	100001
ori	001101	lhu	100101
xor	100110	lw	100011
xori	001110	sb	101000
sll	000000	sh	101001
sllv	000100	sw	101011
sra	000011	mflo	010010

MIPS instruction types

- **R-type:**



- **I-type:**



- **J-type:**



Example

```
00100000 11010001 00000000 00100110
```

- Opcode 001000 → I-type → addi
- rs = 6
- **rt = 17**
- **Immediate = 38**

→ Add the value 38 to register 6 and store the result in register 17 (**addi \$17, \$6, 38**)

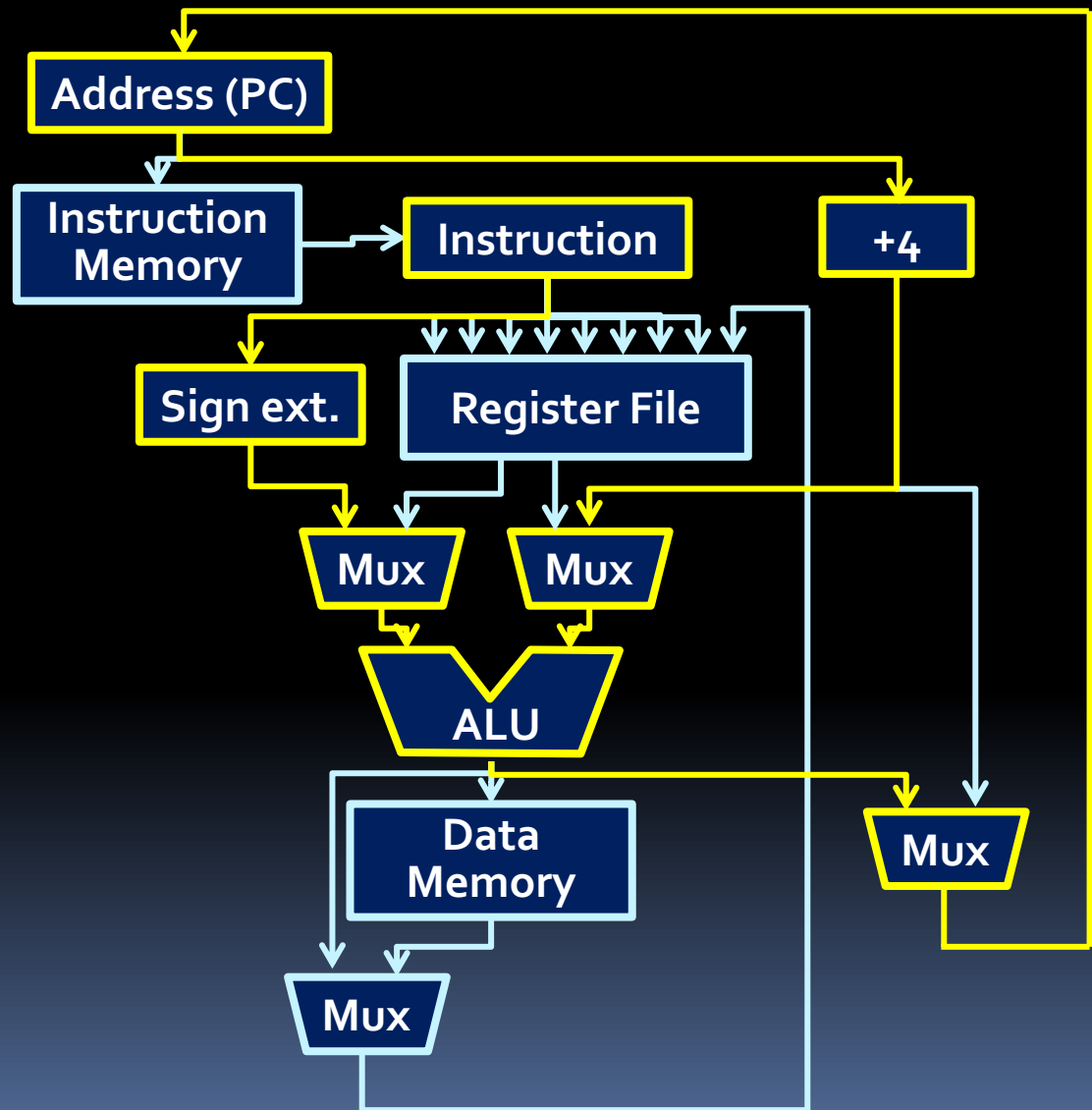
J-type instructions



- Only two J-type instructions:
 - jump (j)
 - jump and link (jal)
- These instructions use the 26-bit coded address field to specify the target of the jump.
- But 32 bits are needed for an address.
 - The first four bits of the destination address stay the same as in the current PC.
 - The bits in positions 27 to 2 in the address are the 26 bits provided in the instruction.
 - The bits at positions 1 and 0 are set to zero (word alignment).

J-type instruction datapath

- Jump and branch use the datapath in similar but different ways:
- Branch calculates new PC value as old PC value + offset.
- Jump loads an immediate value over top of the old PC value.



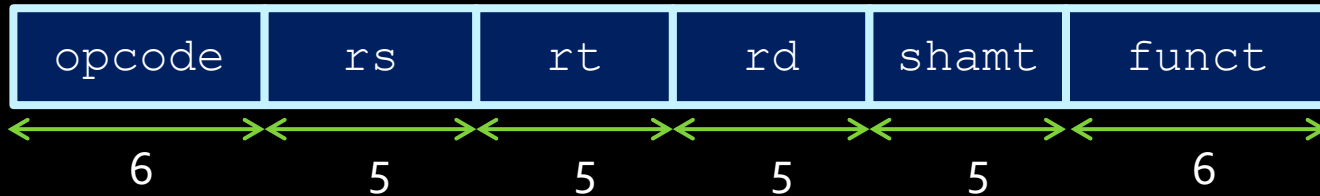
Examples

```
00001010 11010001 00000000 00100110
```

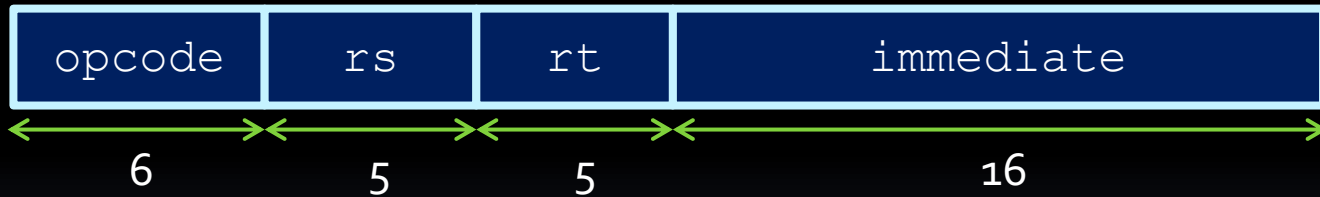

<u>Instruction</u>	<u>Op/Func</u>	<u>Instruction</u>	<u>Op/Func</u>
add	100000	srav	000111
addu	100001	srl	000010
addi	001000	srlv	000110
addiu	001001	beq	000100
div	011010	bgtz	000111
divu	011011	blez	000110
mult	011000	bne	000101
multu	011001	j	000010
sub	100010	jal	000011
subu	100011	jalr	001001
and	100100	jr	001000
andi	001100	lb	100000
nor	100111	lbu	100100
or	100101	lh	100001
ori	001101	lhu	100101
xor	100110	lw	100011
xori	001110	sb	101000
sll	000000	sh	101001
sllv	000100	sw	101011
sra	000011	mflo	010010

MIPS instruction types

- **R-type:**



- **I-type:**



- **J-type:**



Examples

```
00001010 11010001 00000000 00100110
```

- Opcode 000010 → J-type → j
- Address = **10 1101 0001 0000 0000 0010 0110**

→ Jump to address:

```
xxxx10110 1000100 00000000 10011000
```

(xxxx are the current 4 high bits of the PC)

(in assembly: **j 0x2D10026**)

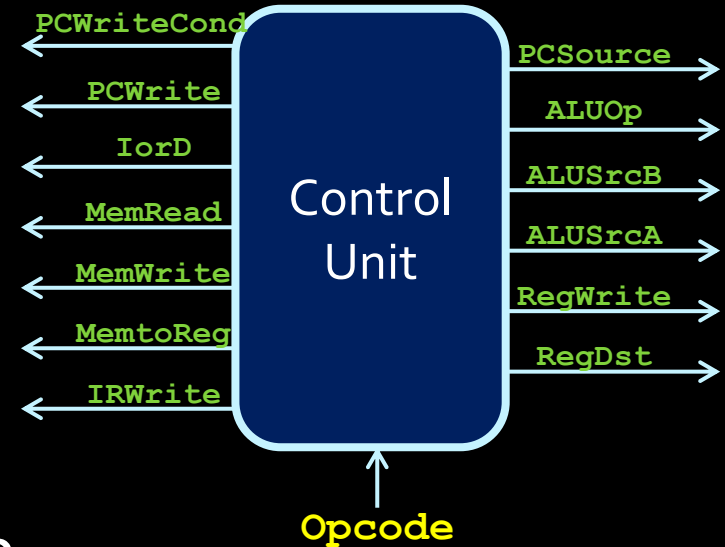
Datapath control

- These instructions are executed by turning various parts of the datapath on and off, to direct the flow of data from the correct source to the correct destination.
- What tells the processor to turn on these various components at the correct times?

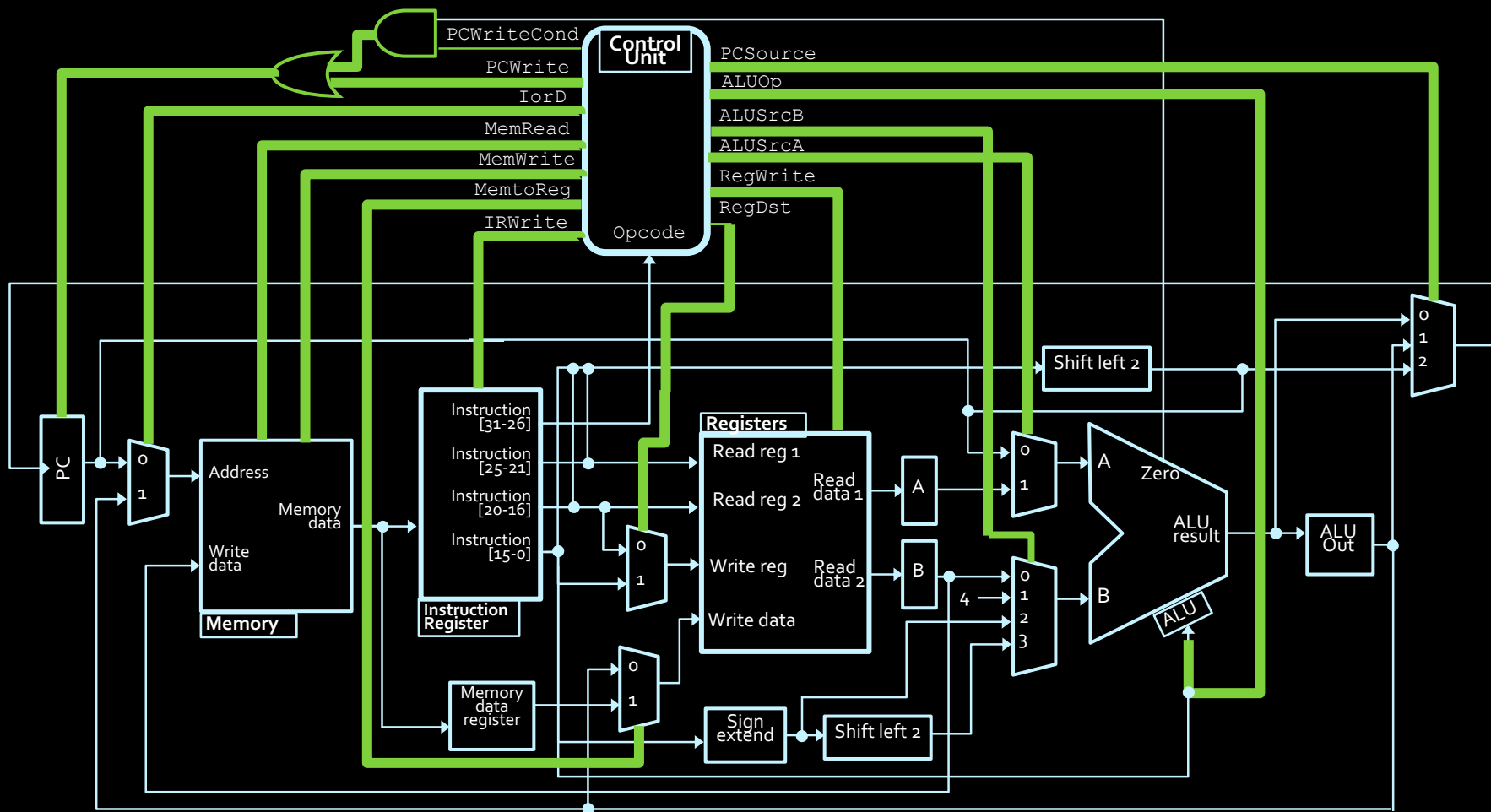


Control unit

- The control unit takes in the **opcode** from the current instruction, and sends **signals** to the rest of the processor.
- Within the control unit is a finite state machine that can occupy multiple clock cycles for a single instruction.
 - The control unit send out different signals on each clock cycle, to make the overall operation happen.

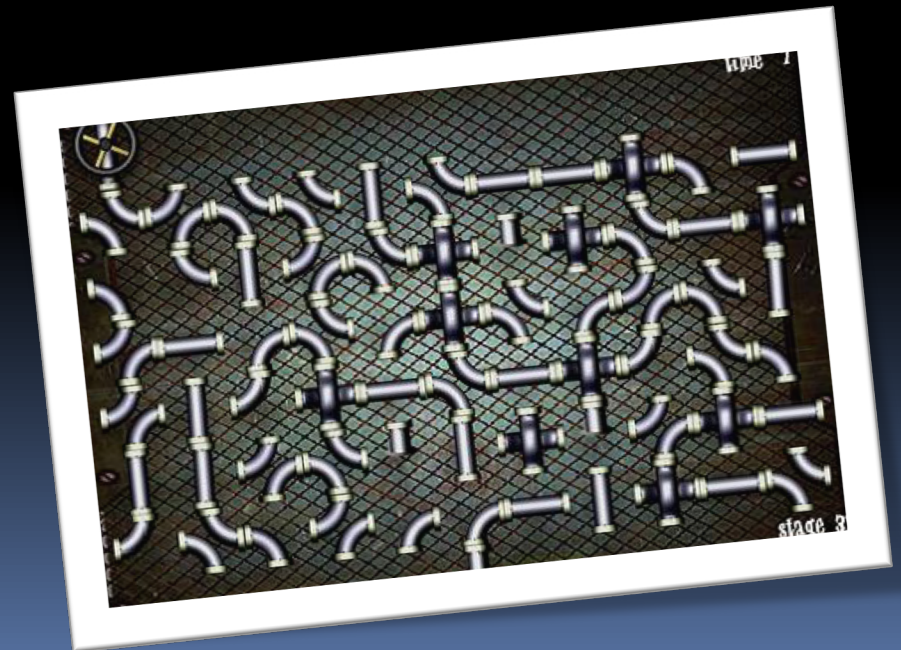


- The control unit sends signals (green lines) to various processor components to enact all possible operations.



Signals → instructions

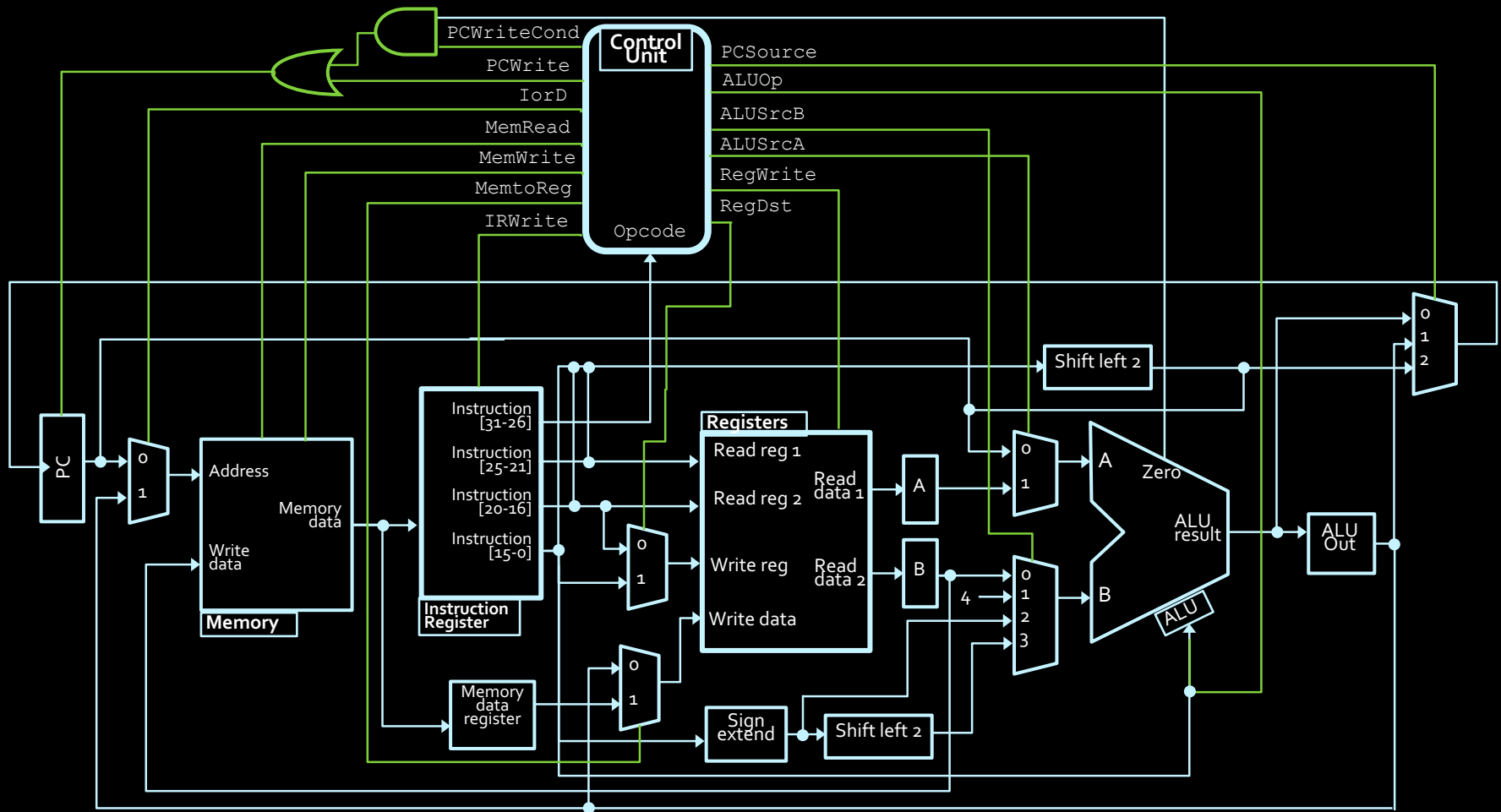
- A certain combination of signals will make data flow from some source to some destination.
- Just need to figure out what signals produce what behaviour.



Control unit signals

- **PCWrite**: Write the ALU output to the PC.
- **PCWriteCond**: Write the ALU output to the PC, only if the Zero condition has been met.
- **IorD**: For memory access; short for “Instruction or Data”. Signals whether the memory address is being provided by the PC (for instructions) or an ALU operation (for data).
- **MemRead**: The processor is reading from memory.
- **MemWrite**: The processor is writing to memory.
- **MemToReg**: The register file is receiving data from memory, not from the ALU output.
- **IRWrite**: The instruction register is being filled with a new instruction from memory.

- The control unit sends signals (green lines) to various processor components to enact all possible operations.



More control unit signals

- **PCSource**: Signals whether the value of the PC resulting from an jump, or an ALU operation.
- **ALUOp** (3 wires): Signals the execution of an ALU operation.
- **ALUSrcA**: Input A into the ALU is coming from the PC (value=0) or the register file (value=1).
- **ALUSrcB** (2 wires): Input B into the ALU is coming from the register file (value=0), a constant value of 4 (value=1), the instruction register (value=2), or the shifted instruction register (value=3).
- **RegWrite**: The processor is writing to the register file.
- **RegDst**: Which part of the instruction is providing the destination address for a register write (*rt* versus *rd*).

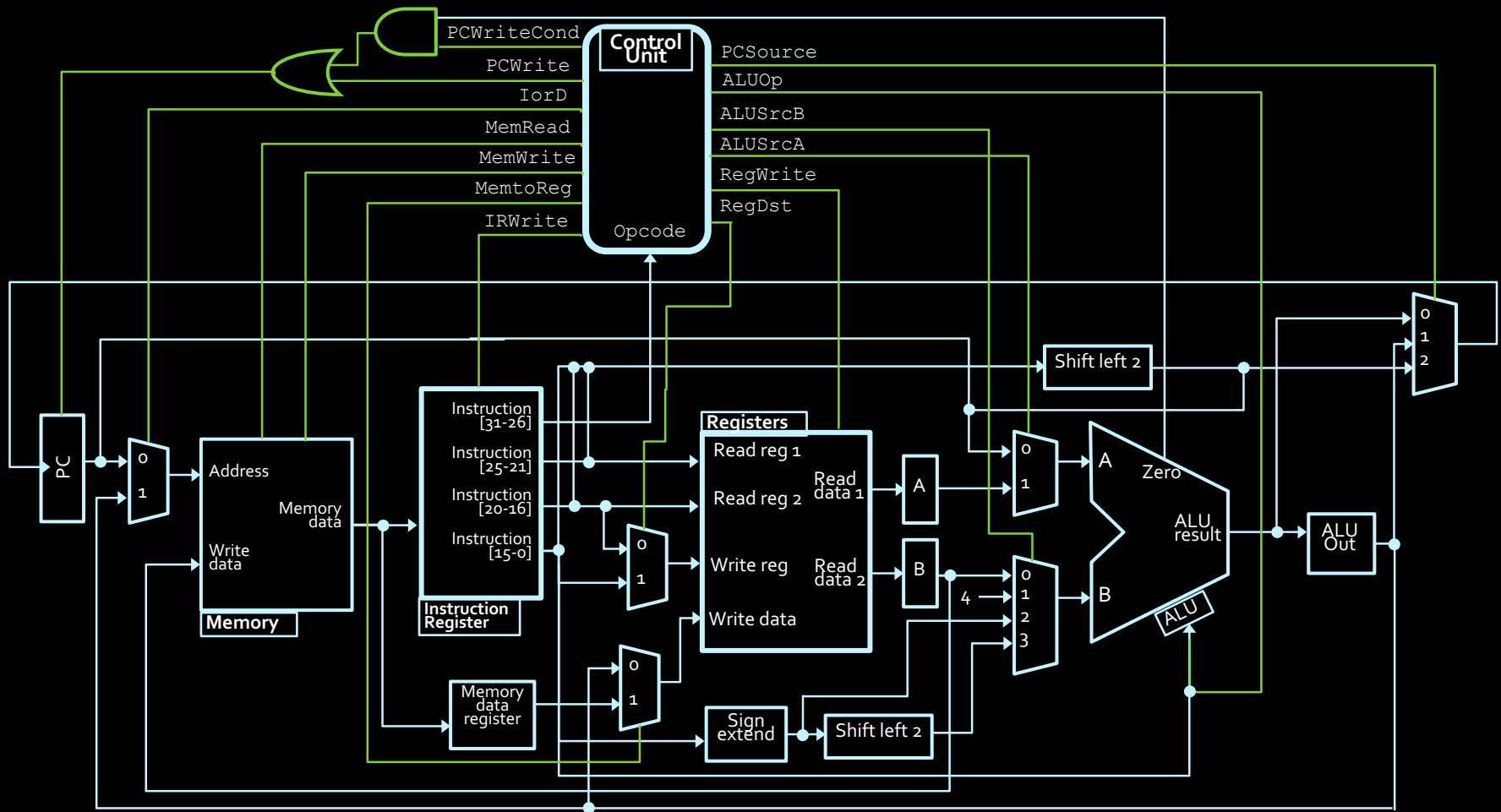
Example instruction



▪ `addi $t7, $t0, 42`

- PCWrite = ?
- PCWriteCond = ?
- IorD = ?
- MemWrite = ?
- MemRead = ?
- MemToReg = ?
- IRWrite = ?
- PCSource = ?
- ALUOp = ?
- ALUSrcA = ?
- ALUSrcB = ?
- RegWrite = ?
- RegDst = ?

- The control unit sends signals (green lines) to various processor components to enact all possible operations.



Example instruction

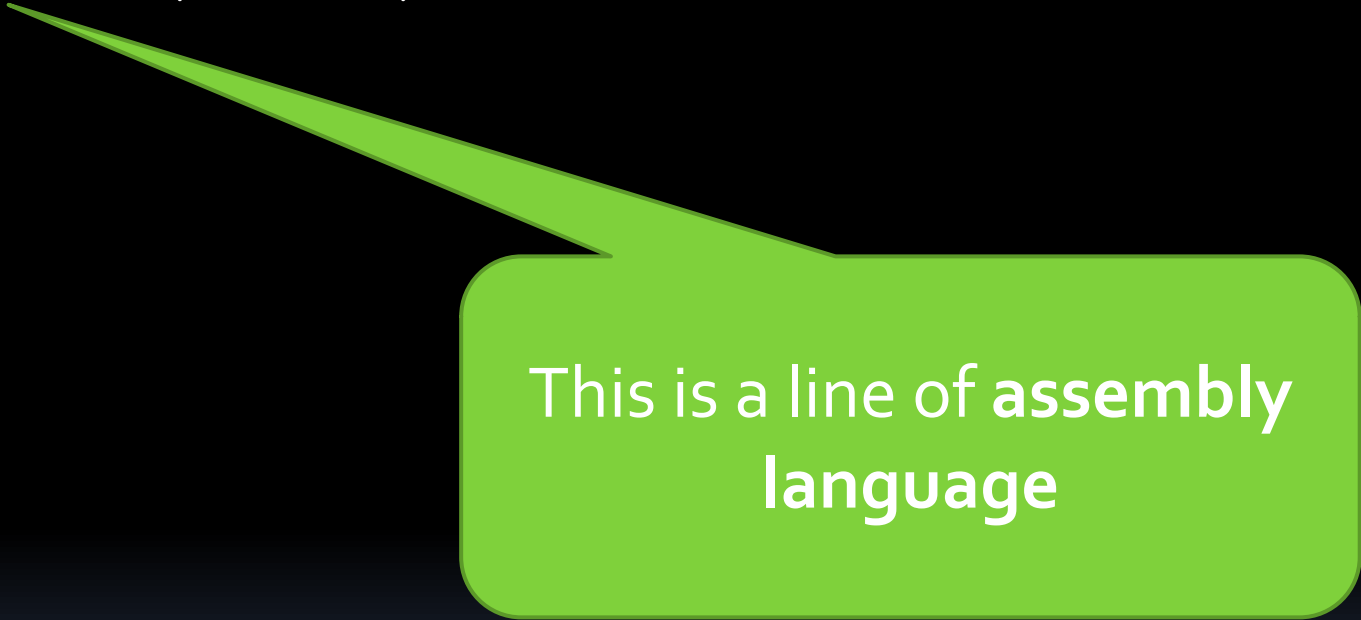


▪ `addi $t7, $t0, 42`

- `PCWrite = 0`
- `PCWriteCond = 0`
- `IorD = X`
- `MemWrite = 0`
- `MemRead = 0`
- `MemToReg = 0`
- `IRWrite = 0`
- `PCSource = X`
- `ALUOp = 001 (add)`
- `ALUSrcA = 1`
- `ALUSrcB = 10`
- `RegWrite = 1`
- `RegDst = 0`

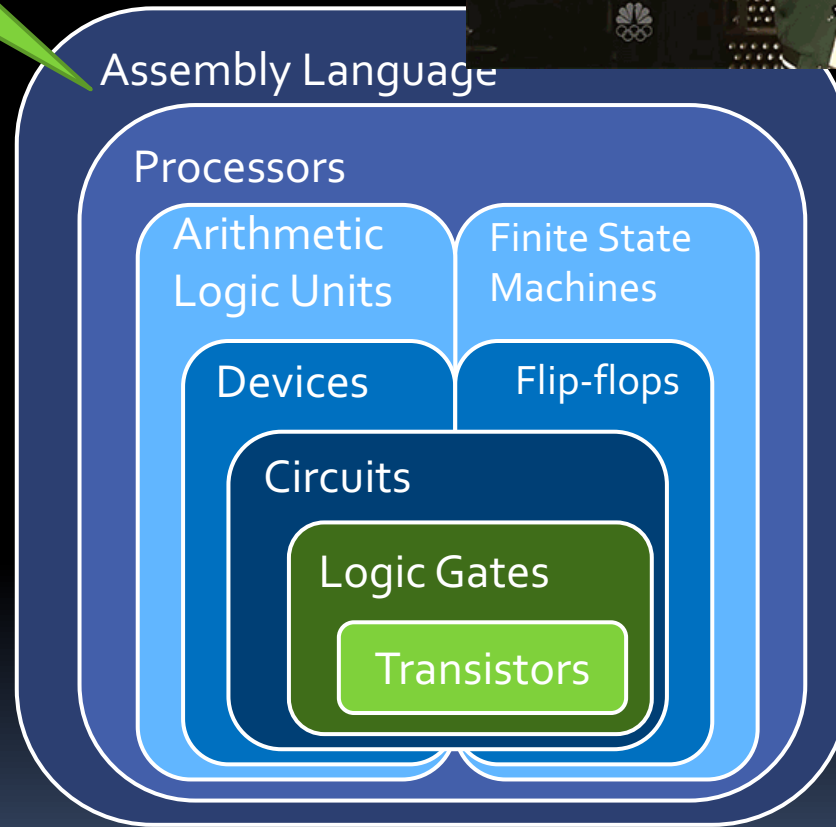
Example instruction

- `addi $t7, $t0, 42`



This is a line of **assembly language**

Started from the
bottom now
we're here



Tale of a program

- User writes code
- Compile code into machine level instructions
- Save instructions in an executable file
- Run the executable file
 - Load file into memory
 - Set PC
 - CPU loads instructions into instruction register
 - Control unit reads op-code
 - Signals turn on/off
 - Billions of transistors turning on/off
 - Trillions of electrons start flowing

Hello, World