Week 7 Review

Let’s test our understanding

Try these questions first:

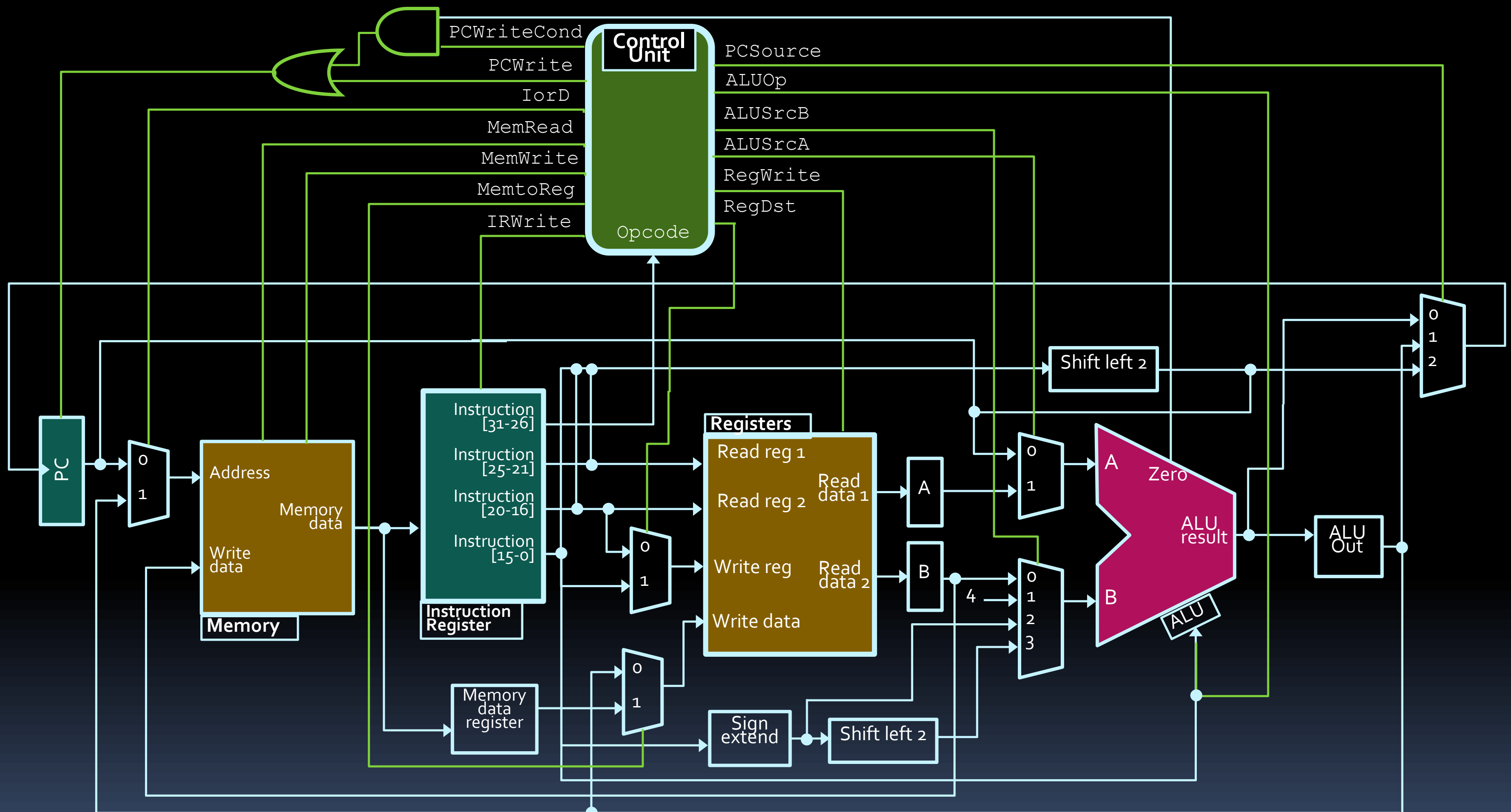
1. Where are instructions stored?
2. How long is a single instruction?
3. What is the role of the Program Counter (PC)?
4. What do we mean by “instruction fetch”?
5. Where does the processor keep the instruction that is currently being executed?

Let’s test our understanding

1. Where are instructions stored?
   - In memory, along with the input data values
2. How long is a single instruction?
   - 4 bytes (32 bits)
3. What is the role of the Program Counter (PC)?
   - Store the location (address) of the current instruction.
4. What do we mean by “instruction fetch”?
   - Retrieve an instruction from memory.
5. Where does the processor keep the instruction that is currently being executed?
   - In the Instruction Register.

Storage

- Register file: small and fast
  - Used as temporary / scratch space for computations.
- Memory: large but slower
  - Stores program instructions and data
- Single registers like PC and instruction register
  - Used for controlling the execution process
- Datapath connects these to ALU.
- Control unit controls the data path.



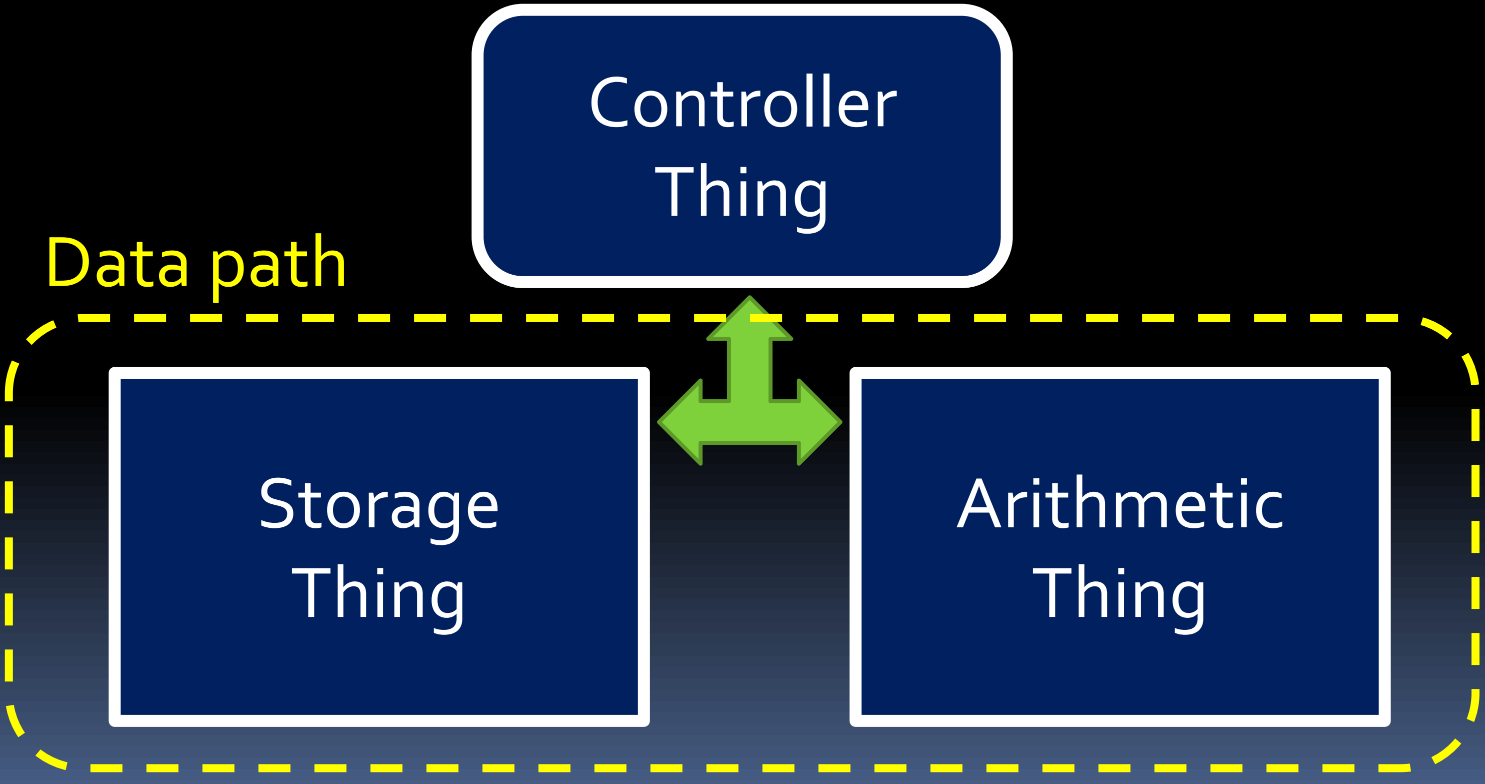
PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Control Unit
Opcode
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0
1
Address
Memory data
Write data
Memory
Instruction [31-26]
Instruction [25-21]
Instruction [20-16]
Instruction [15-0]
Instruction Register
Memory data register
Registers
Read reg 1
Read reg 2
Write reg
Write data
Read data 1
Read data 2
A
B
4
0
1
2
3
Sign extend
Shift left 2
Shift left 2
A
B
Zero
ALU result
ALU
ALU Out
0
1
2

Question

- Your RAM unit has 6 address bits going into it. Given a 32-bit architecture, how many integers (words) is your RAM unit able to store?

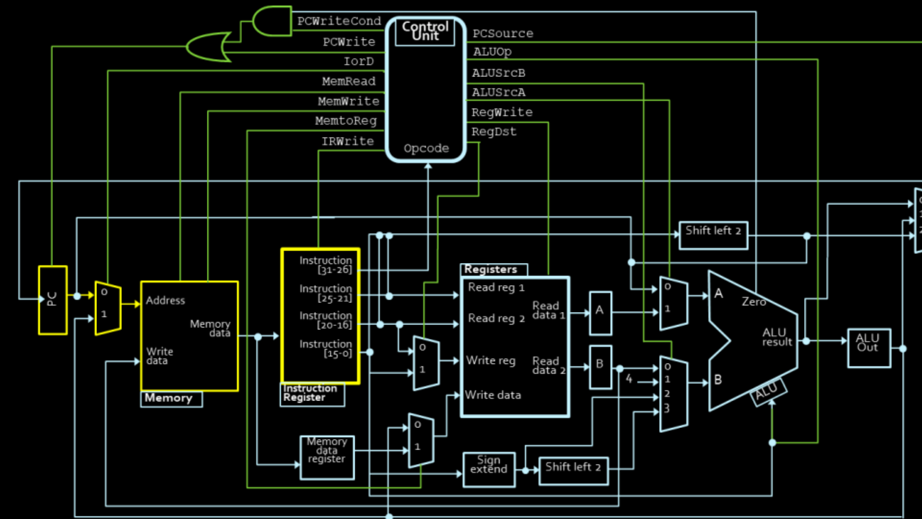
- Be careful here!
  - 6 address bits → 2^6 memory slots = 64 bytes.
  - 32-bit architecture → 4 bytes per integer.
  - RAM capacity = 64 / 4 = 16 integers in memory.

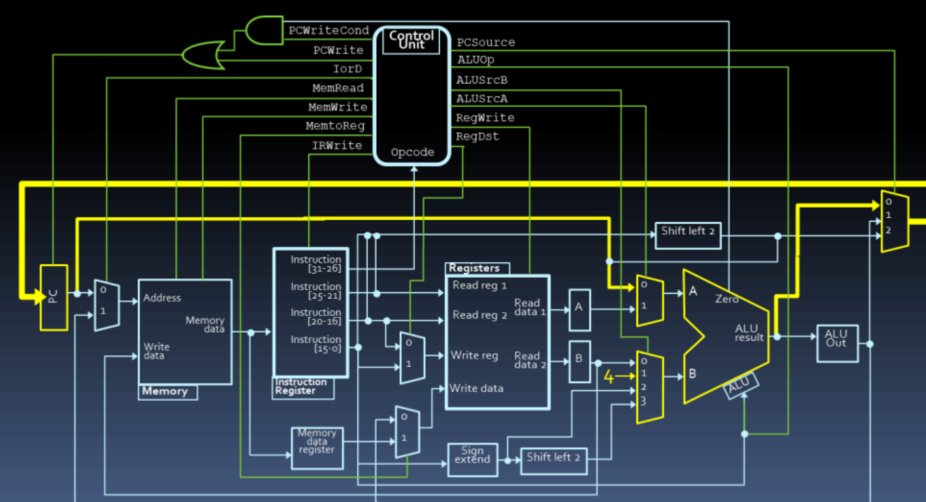
- We learned what the Arithmetic and Storage Things do
- And we can build a data path



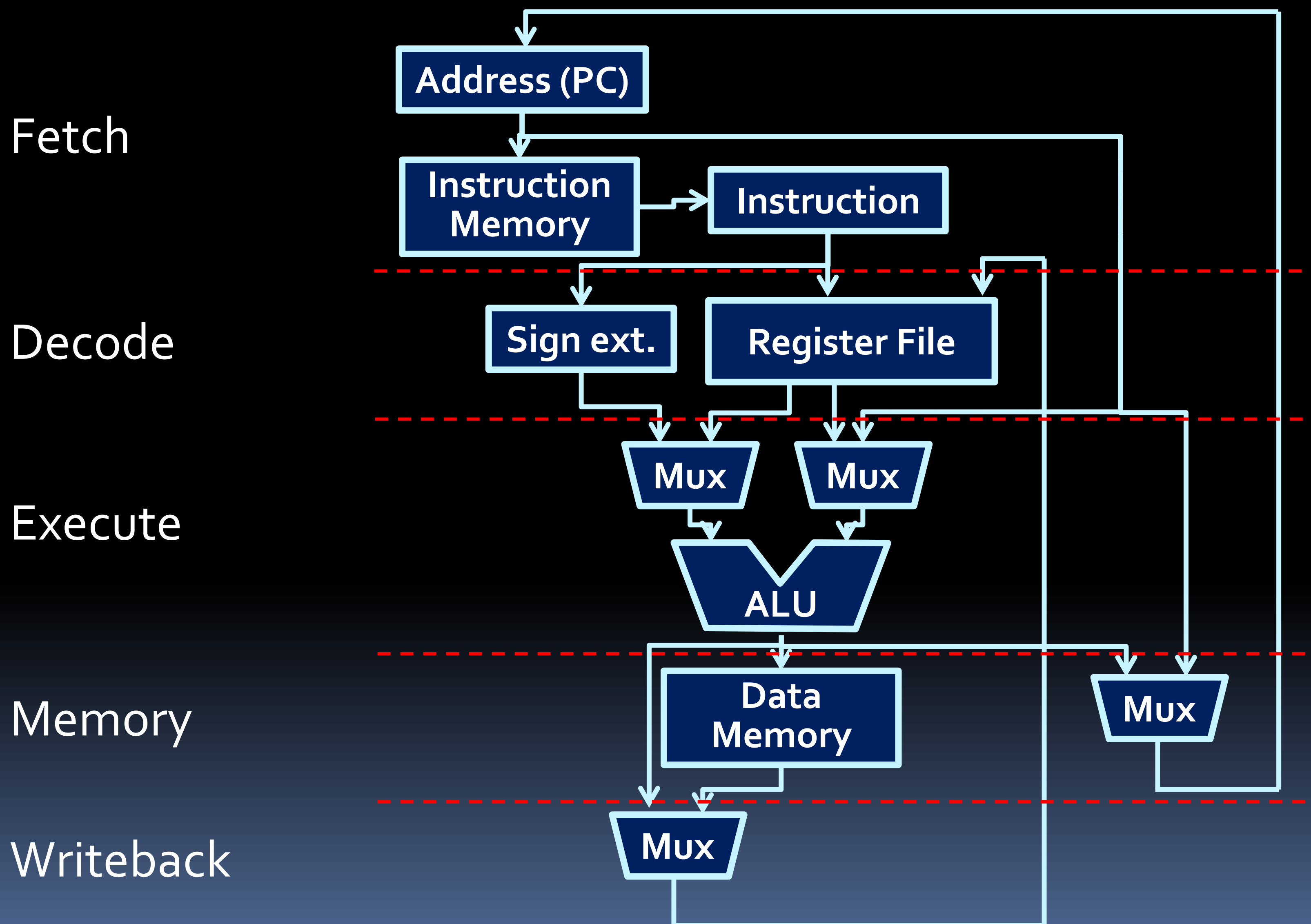
Control Unit

1. **Fetch** instruction from memory into instruction register
2. **Decode and execute** instruction.
   - Execution time can depend on instruction.
3. **Advance** to next instruction using on of:
   - PC ← PC + 4 or
   - PC ← from ALU





“Linear” Datapath



MIPS instruction types

- R-type:

opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

- I-type:

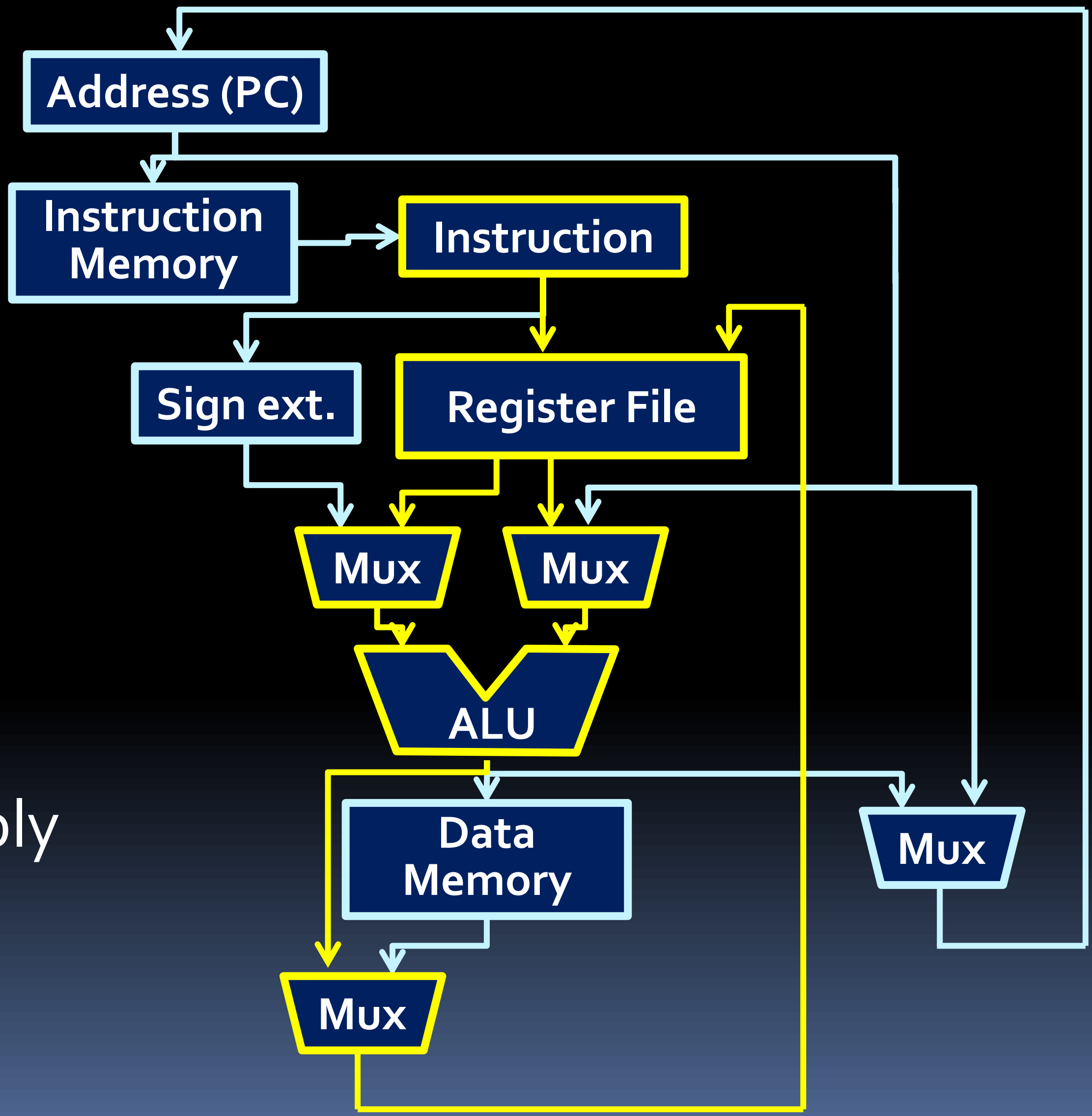
opcode	rs	rt	immediate
6	5	5	16

- J-type:

opcode	address
6	26

R-Type

- Rd ← OP(Rs, Rt)
- Sources: 2 reg.s
- Dest: reg.
- Examples:
  - add, subtract, multiply
  - shift left, shift right
  - and, not, xor, or,…



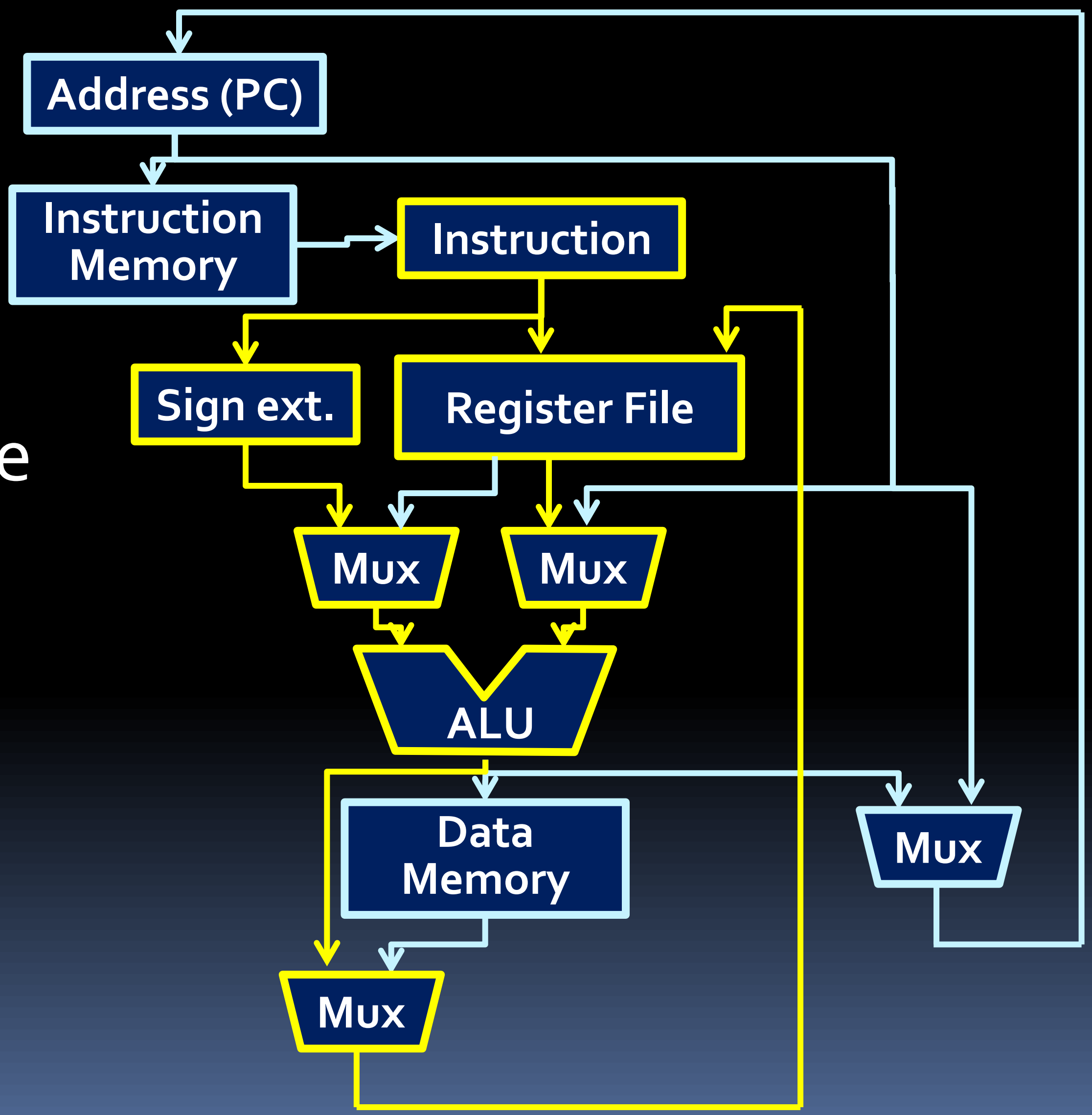
I-type instructions

opcode	rs	rt	immediate
6	5	5	16

- I-type instructions have a 16-bit immediate field.
- This field is a constant value, which is used for:
  - Arithmetic or other operations:
    - e.g., to compute $rt = $rs + <immediate>
  - a displacement (offset) for memory address.
    - e.g, write to memory address $rs + <immediate>
  - a branch target offset to jump to another instruction
    - e.g., if $rt == $rs, branch to $pc + <immediate>

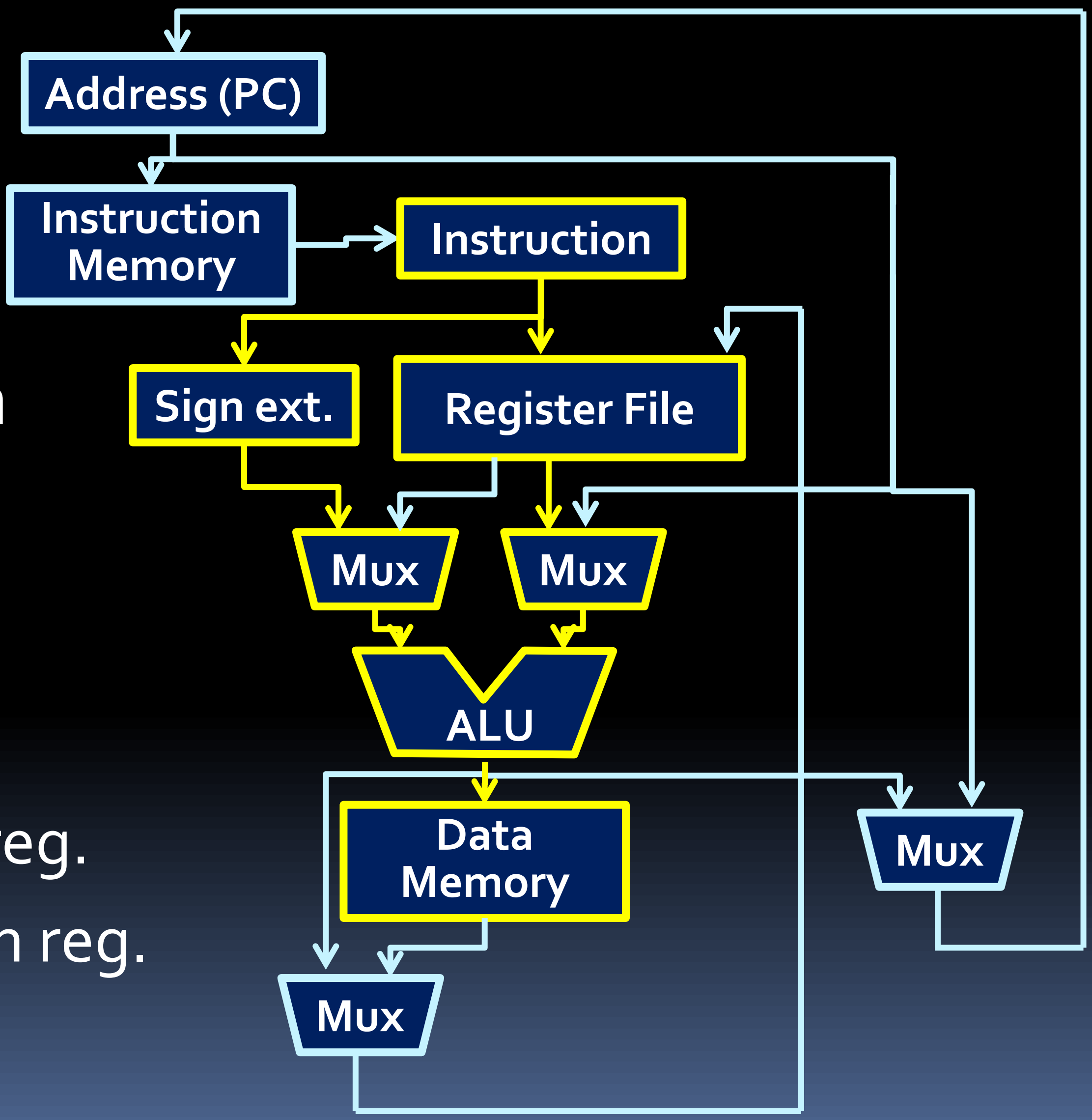
I-Type #1

- Rt ← OP(Rs, **IM**)
- Source 1: reg.
- Source 2: immediate (instruction reg.)
- Dest: reg.
- Examples:
  - addi, andi



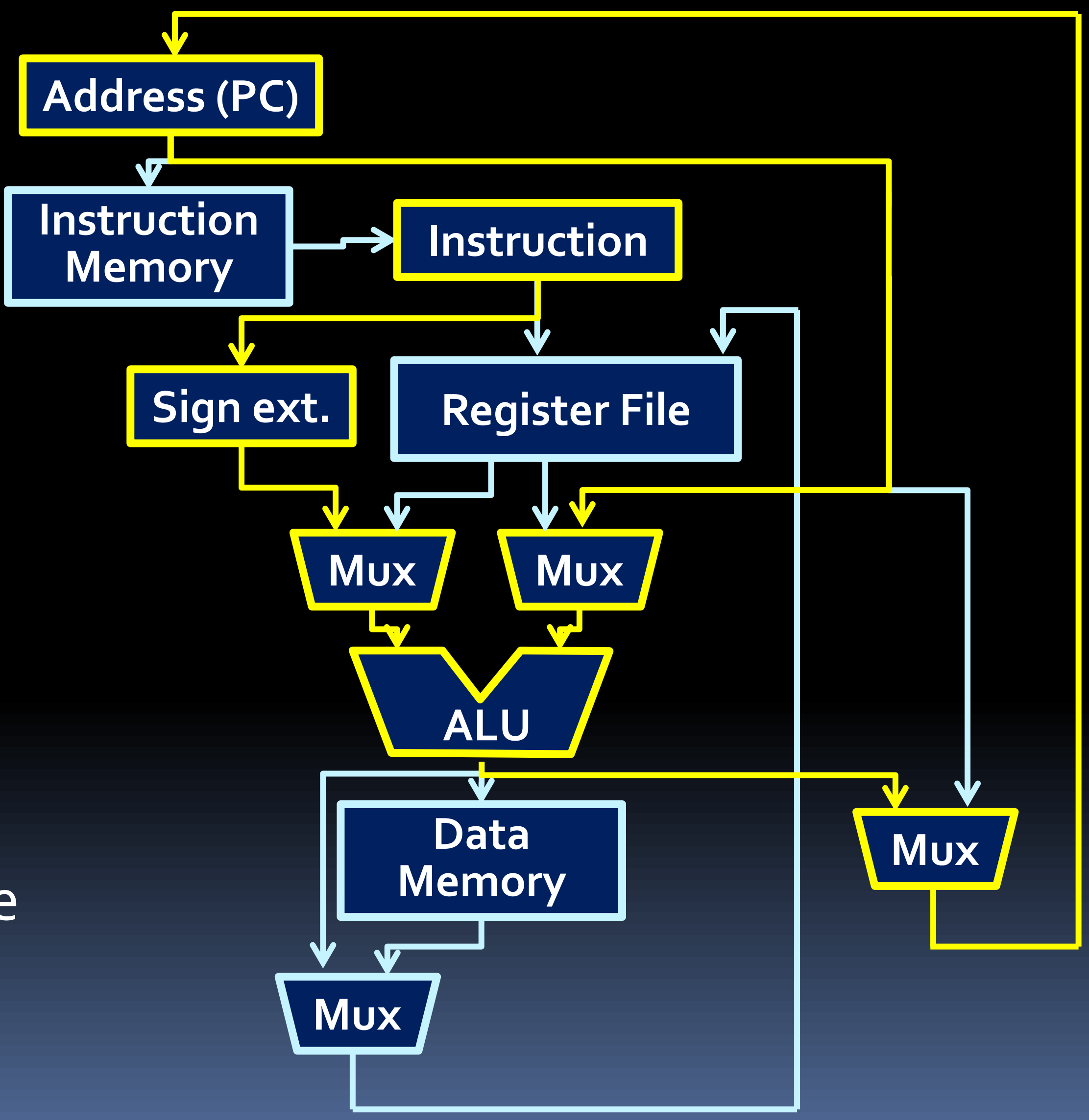
I-Type #2

- Load/Store
- Source: reg or mem
- Dest: mem or reg.
- Examples:
  - lw - load value into reg.
  - sw - store value from reg.



I-Type #3

- Branch:
  PC ← PC + IM
- Source 1: PC
- Source 2: Immediate
- Dest: PC
- Examples:
  - Branch if register value is equal to 0

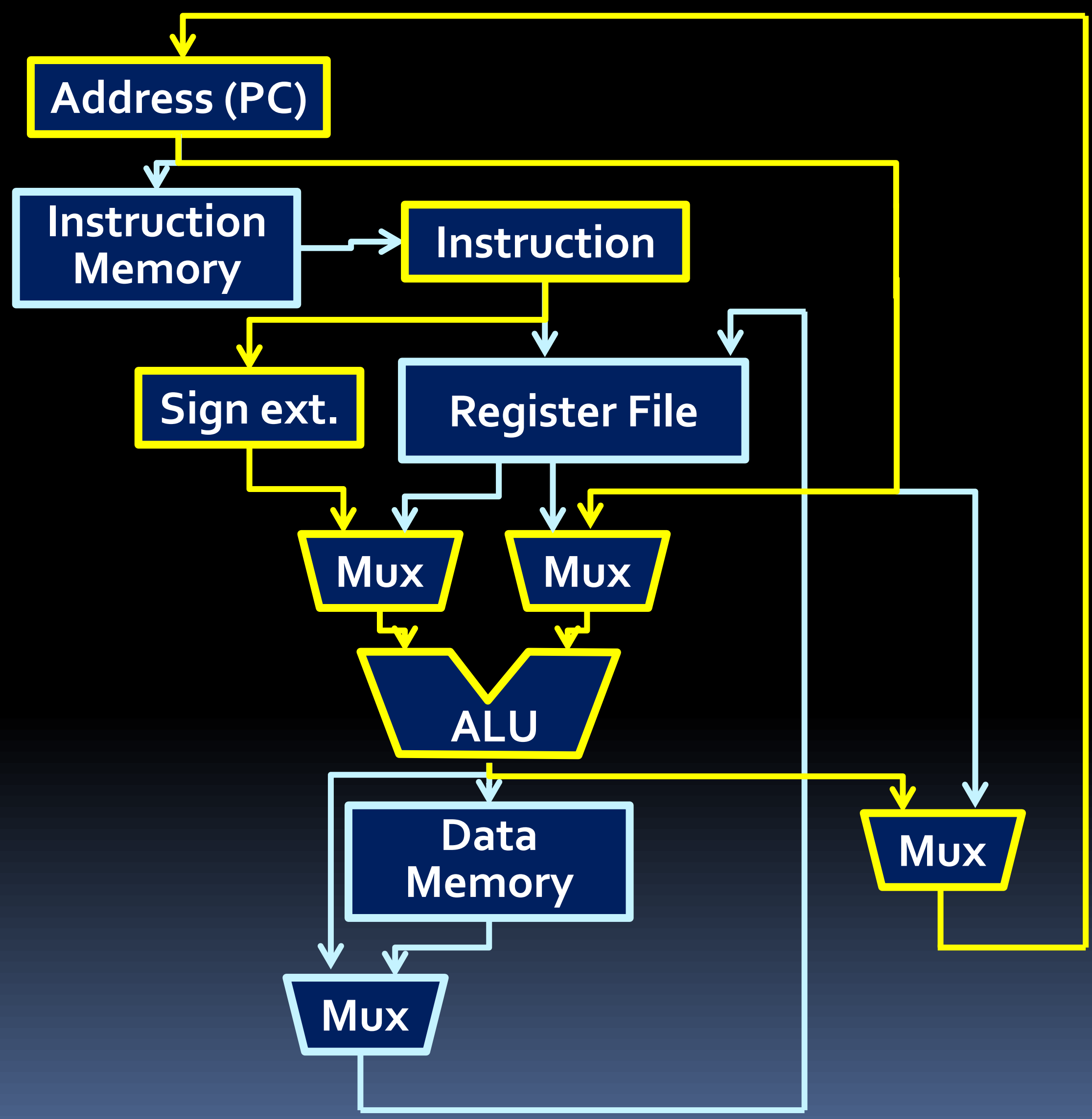


I-type: Branching

- Immediate field is only 16 bits.
  - We cannot store the address of the destination since it is 32 bits.
- In practice, branch targets tends to be close to origin.
- Store the signed difference between the address of current instruction (the current PC) and the address of the target instruction.
- Also, since instructions are 4 bytes and word-aligned, we this difference is a multiple of 4.
  - Don’t store the lowest 2 bits of the difference – they are zero anyway! Use the extra space to store higher bits.
- So we actually encode a 18-bit signed difference

J-type

- Jump:
  PC ← address
- Source 1: PC
- Source 2: Address
- Dest: PC
- Examples:
  - j , jal



Decoding Instructions

1. Look at opcode (first 6 bits)

xxxxxx	??????????????????????????

2. If 000000
   → R-type

opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

3. Otherwise lookup opcode in table:
   → I-type

opcode	rs	rt	immediate
6	5	5	16

   → J-type

opcode	address
6	26

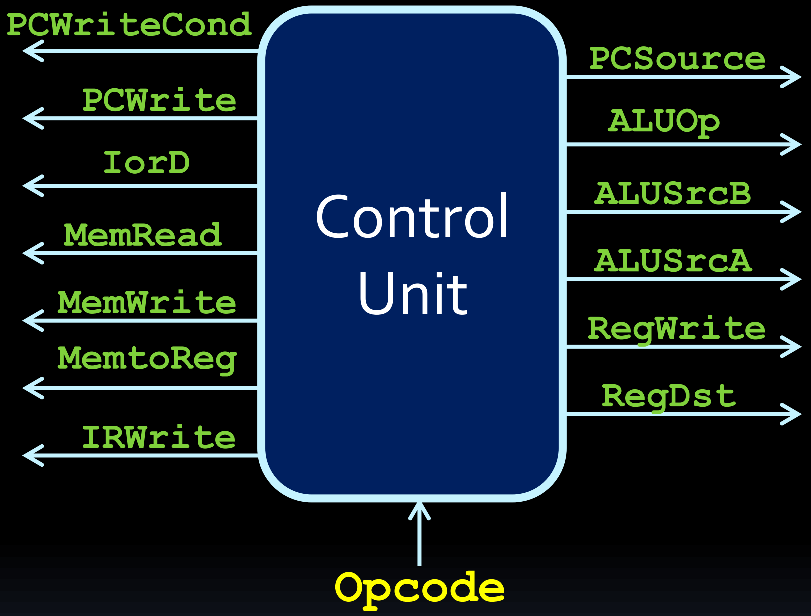
Opcodes

- First 6 bits of the instruction.
- I-type: white
- J-type: pink
  - Only `j`, `jal`.
- R-type opcode is `000000`
  - Instead we list the function (last 6 bits of instruction)
  - Marked yellow

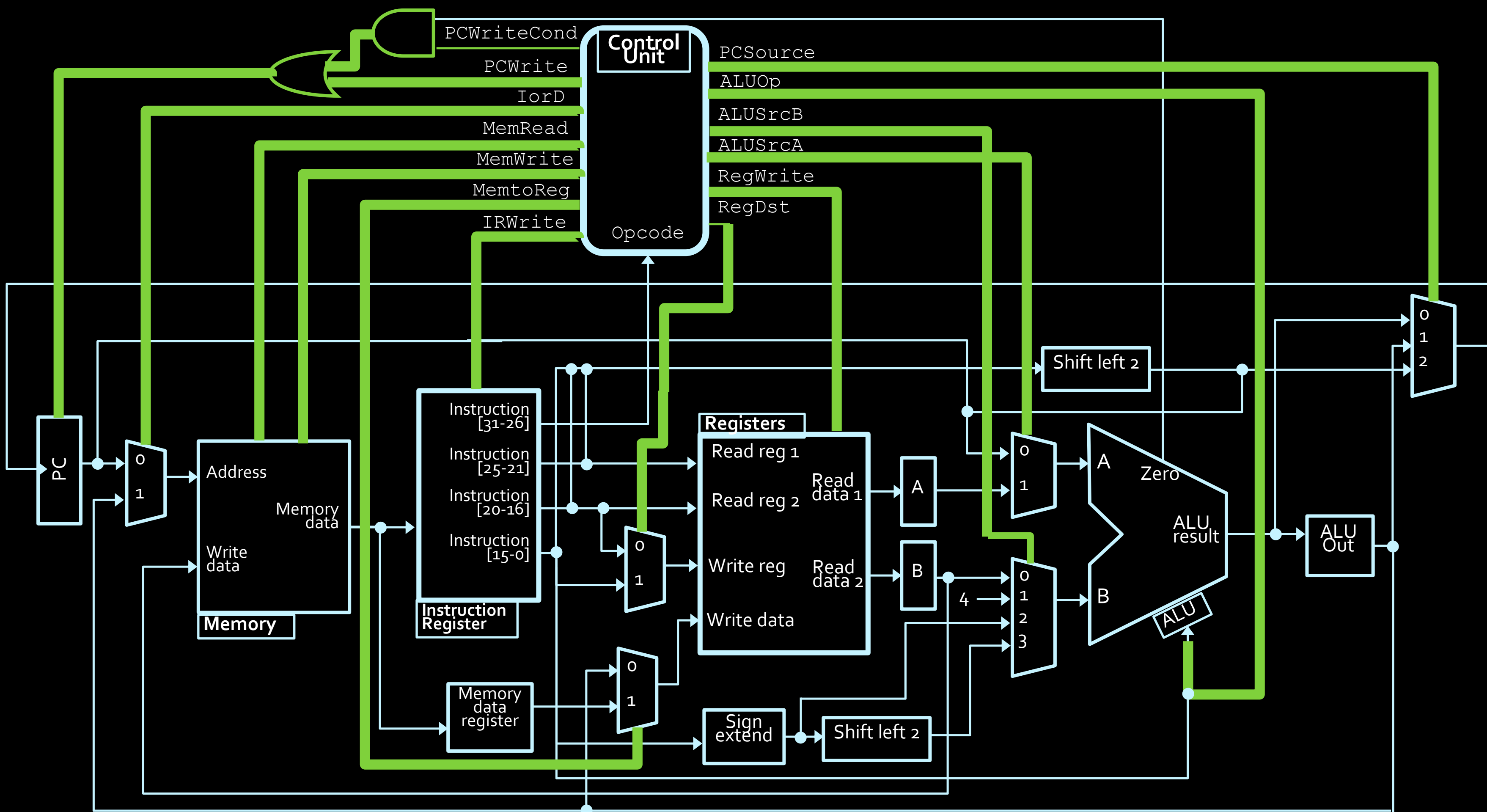
Instruction	Type	Op/Func	Instruction	Type	Op/Func
add	R	100000	srav	R	000111
addu	R	100001	srl	R	000010
addi	I	001000	srlv	R	000110
addiu	I	001001	beq	I	000100
div	R	011010	bgtz	I	000111
divu	R	011011	blez	I	000110
mult	R	011000	bne	I	000101
multu	R	011001	j	J	000010
sub	R	100010	jal	J	000011
subu	R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

Control Signals

- The control unit makes computation happen.
- A Finite State Machine.
- Given opcode, send the right combination of control signals to execute the instruction.
  - Each instruction can take multiple cycles

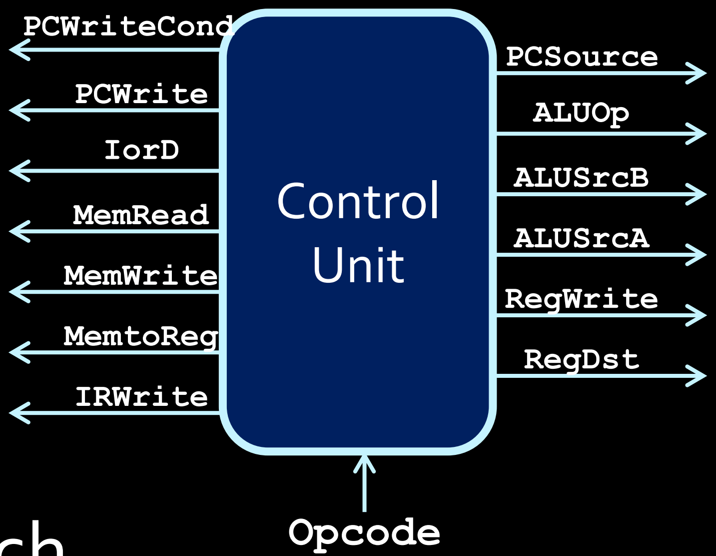


- The control unit sends signals (green lines) to various processor components to enact all possible operations.



Controlling the signals

- Need to understand the role of each signal, and what value they need to have in order to perform the given operation.
- So, what's the best approach to make this happen?



Basic approach to datapath

1. Figure out the data source(s) and destination.
2. Determine the path of the data.
3. Deduce the signal values that cause this path:
   a) Start with `Read` & `Write` signals
      - At most one ___`Write` signal should be high at a time.
   b) Then, mux signals along the data path.
   c) Non-essential signals get an `X` value.

Question

`0000 0000 0110 0101 0100 0000 0010 0111`

- What is the type of this instruction?
- What does it do?
- Which register stores the result?

0000 0000 0110 0101
0100 0000 0010 0111

- What is the type of this instruction?
- What does it do?
- Which register stores the result?

Instruction	Op/Func	Instruction	Op/Func
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:

opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

- I-type:

opcode	rs	rt	immediate
6	5	5	16

- J-type:

opcode	address
6	26

Question

`0000 0000 0110 0101 0100 0000 0010 0111`

- What is the type of this instruction?
- What does it do?
- Which register stores the result?

Question

opcode

`0000 0000 0110 0101 0100 0000 0010 0111`

- What is the type of this instruction?
  - R-type
- What does it do?
- Which register stores the result?

Question

opcode	rs	rt	rd	shamt	funct

0000 0000 0110 0101 0100 0000 0010 0111

- What is the type of this instruction?
  - R-type
- What does it do?
  - `funct = 100111`
- Which register stores the result?

0000 0000 0110 0101
0100 0000 0010 0111

funct = 100111

Instruction	Op/Func	Instruction	Op/Func
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

Question

opcode	rs	rt	rd	shamt	funct

0000 0000 0110 0101 0100 0000 0010 0111

- What is the type of this instruction?
  - R-type
- What does it do?
  - nor
- Which register stores the result?
  - `rd = 01000`
  - register 8 (known as `$t0` in MIPS assembly)

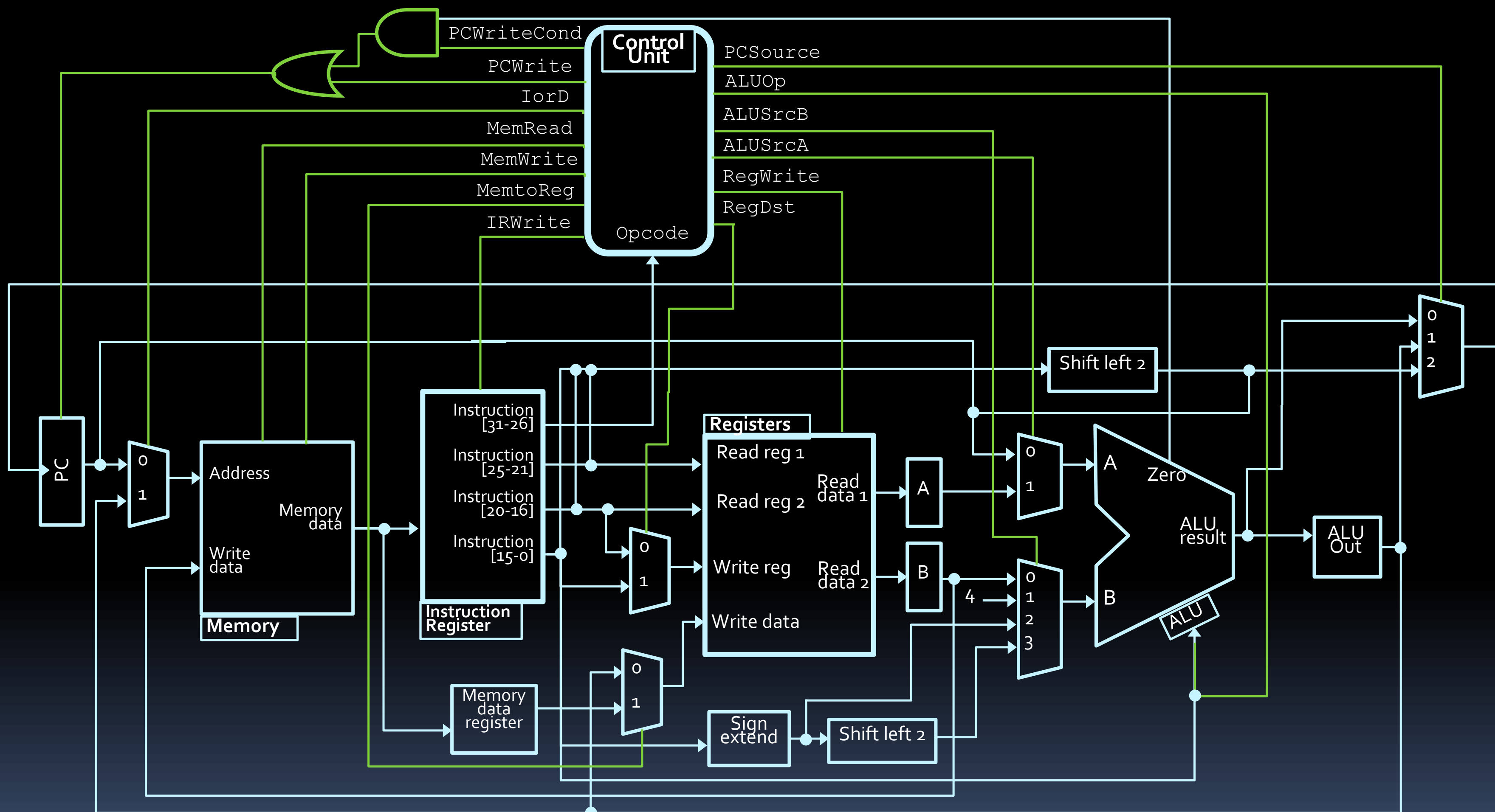
Week 8: Intro to Assembly Programming

```
loop: lw   $t3, 0($t0)
      lw   $t4, 4($t0)
      add  $t2, $t3, $t4
      sw   $t2, 8($t0)
      addi $t0, $t0, 4
      addi $t1, $t1, -1
      bgtz $t1, loop
```

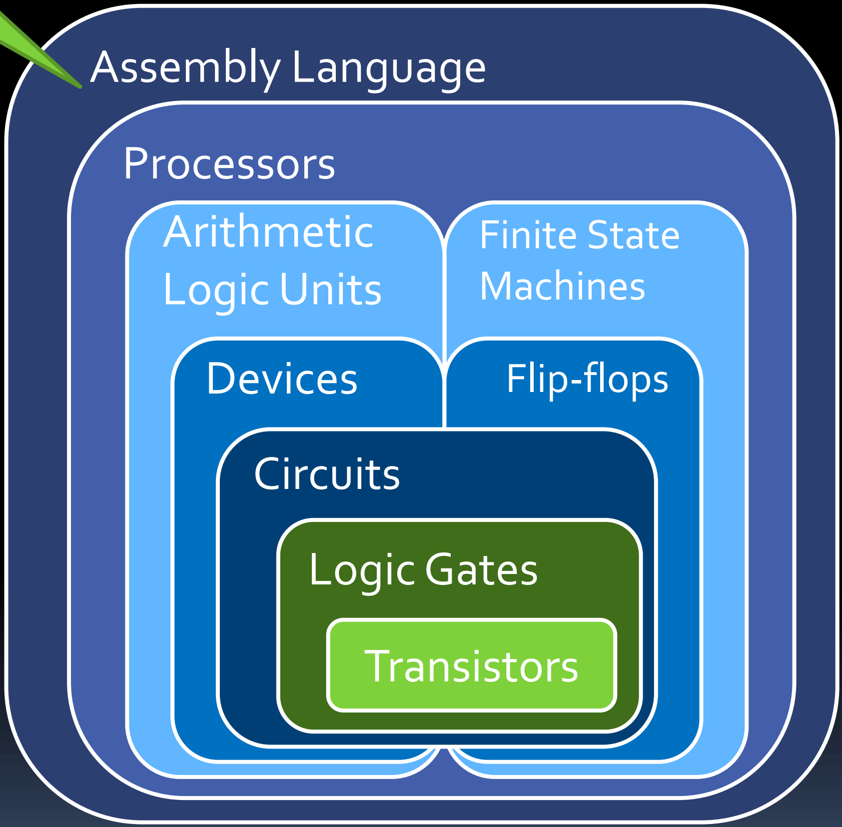
Assembler →

```
0x8d0b0000
0x8d0c0004
0x016c5020
0xad0a0008
0x21080004
0x2129ffff
0x1d20fff9
```

The MIPS Microprocessor



Started from the bottom now we're here



Hello, World

Tale of a program

- Write code
- Compile code into machine code 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

Intro to Machine Code

- Now that we have a processor, operations are performed by:
  - Load instructions one by one to instruction register
  - The control unit reads and decodes instruction according to the opcode in the first 6 bits.
  - The control unit sends a sequence of signals to the datapath.
- A program is just a sequence of instructions in machine cdoe

A program!

- Below is the content of an executable file “mystery.exe”,
- What does this program do?

```
10001110 00001000 01011010 11110001
10001110 00101001 11010010 00110010
00000001 00001001 01010000 00100000
10001110 01001011 11110011 00110111
00000000 00001100 00110001 00000000
00000010 01101010 10100000 00100010
10101101 11010100 00001111 01011010
```

Assembly vs Machine Code

- Machine code is hard to read.
- Represent instructions as user-readable code words.
  - User-friendly machine code: one line ←→ one instruction
  - Each processor architecture has its own version.
- Example: `C = A + B`
  - Assume `A` is stored in `$t1`, `B` in `$t2`, `C` in `$t3`.
  - Assembly language instruction:

```
add $t3, $t1, $t2
```

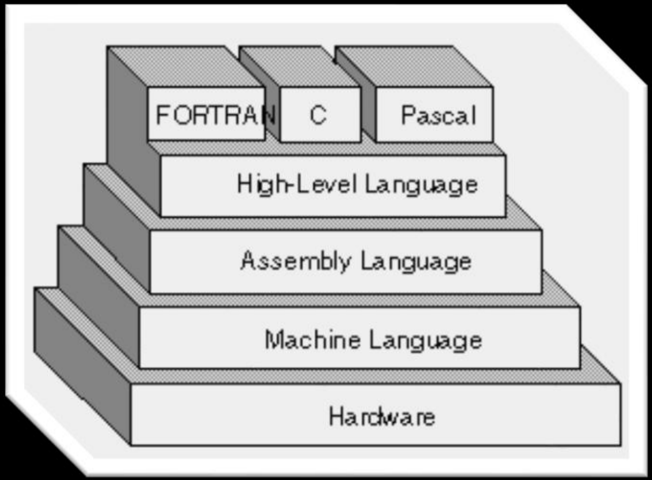
  - Machine code instruction:

```
000000 01001 01010 01011 00000 100000
```

1-to-1 mapping for all assembly code and machine code instructions!

Assembly language

- Assembly language is the lowest-level language that you'll ever program in.



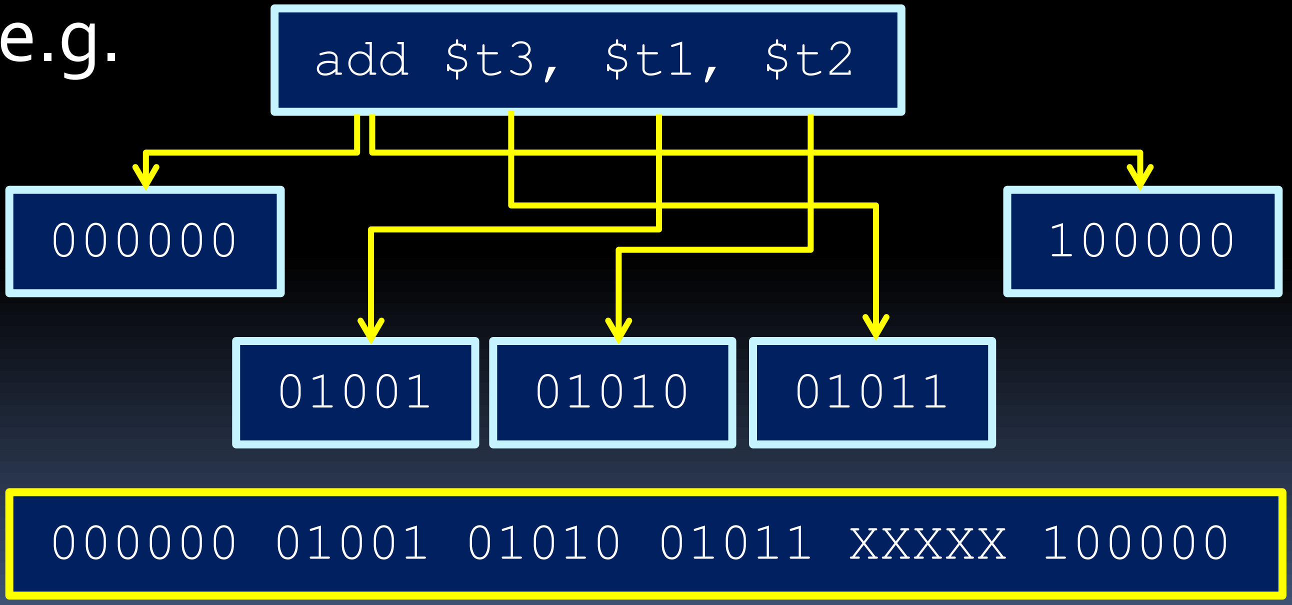
- Many compilers translate their high-level program commands into assembly commands, which are then converted into machine code and used by the processor.
- Note: There are multiple types of assembly language, especially for different architectures!

Why learn assembly?

- Understand how computers really work
- Understand how compilers really work
- Better analyze and debug code
  - Runtime, control flows, pointers, stack overflows
- Appreciate constructs of high-level languages
- Connect your high-level programming knowledge to hardware
- It's on the exam…

Assembly to Machine Code

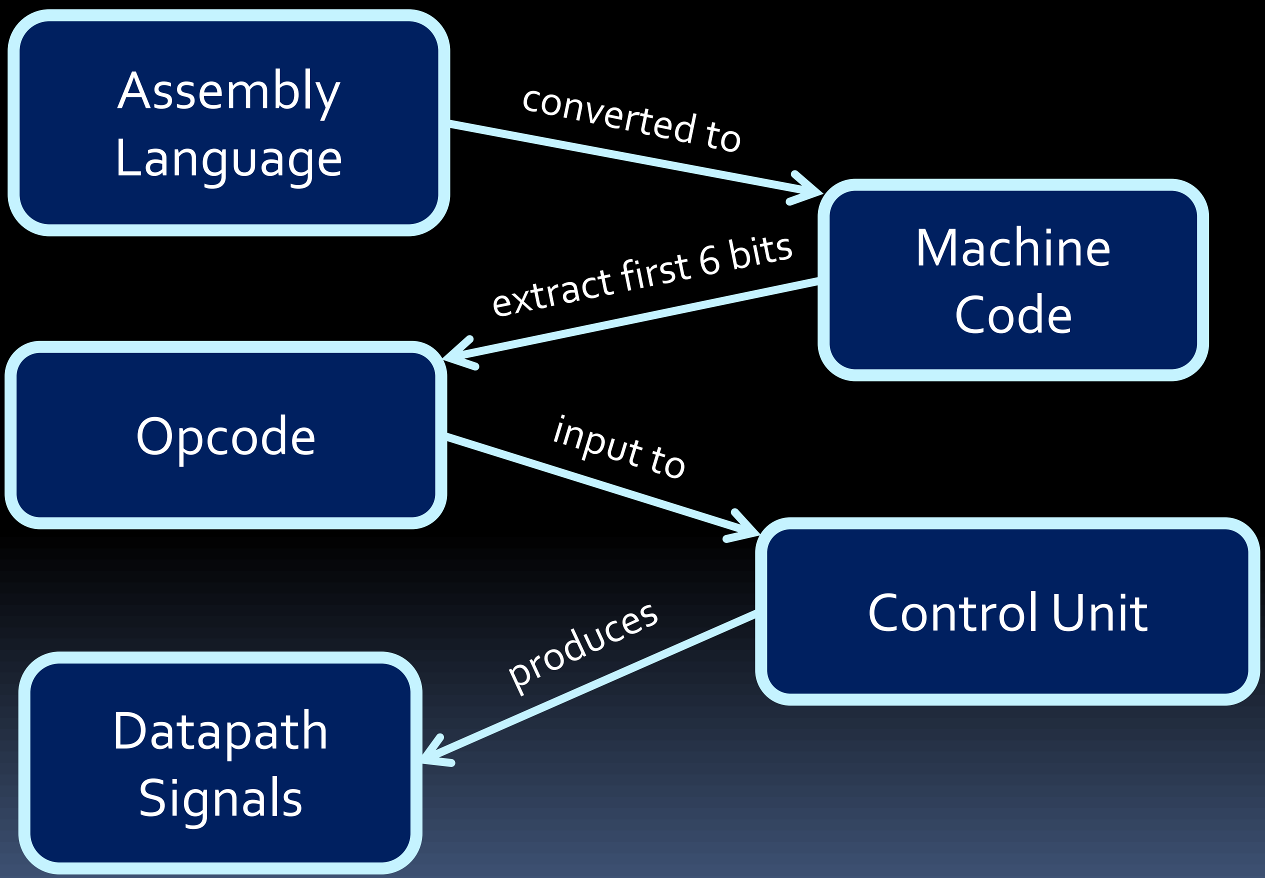
- Encoding is reverse of decoding.
- We need to know how to encode the operation to perform, and the register values to operate on.
- e.g.



Assembler

- The program that converts assembly code to executable is NOT called a compiler.
- It's called an assembler, because there is no fancy complication needed, it just assembles the lines!
- Try to write a simple one for the processor you implement in Lab 4

How things fit together



MIPS Assembly

```
00000000  01 00 FF FF 00 00 00 00  00 00 00 00 40 00 CC 80  ............@..
00000010  0C 00 00 00 00 00 26 01  8F 00 00 00 00 00 53 00  ......&.......S.
00000020  65 00 6C 00 65 00 63 00  74 00 20 00 52 00 75 00  e.l.e.c.t. .R.u.
00000030  6C 00 65 00 00 00 08 00  00 00 00 01 4D 00 53 00  l.e.........M.S.
00000040  20 00 53 00 68 00 65 00  6C 00 6C 00 20 00 44 00   .S.h.e.l.l. .D.
00000050  6C 00 67 00 00 00 00 00  00 00 00 00 00 02 00 00  l.g.............
00000060  03 01 A1 50 53 00 3A 00  C3 00 36 00 32 25 00 00  ...PS.:...6.2%..
00000070  FF FF 83 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080  03 00 01 50 0E 00 56 00  41 00 0A 00 4A 26 00 00  ...P..V.A...J&..
00000090  FF FF 80 00 26 00 41 00  70 00 70 00 6C 00 79 00  ....&.A.p.p.l.y.
000000a0  20 00 74 00 6F 00 20 00  61 00 6C 00 6C 00 00 00   .t.o. .a.l.l...
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 01 00 01 50  ...............P
000000c0  7E 00 7D 00 32 00 0E 00  01 00 00 00 FF FF 80 00  ~.}.2...........
000000d0  4F 00 4B 00 00 00 00 00  00 00 00 00 00 00 00 00  O.K.............
000000e0  00 00 01 50 B4 00 7D 00  32 00 0E 00 02 00 00 00  ...P..}.2.......
000000f0  FF FF 80 00 43 00 61 00  6E 00 63 00 65 00 6C 00  ....C.a.n.c.e.l.
00000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 50  ...............P
00000110  EA 00 7D 00 32 00 0E 00  09 00 00 00 FF FF 80 00  ..}.2...........
00000120  26 00 48 00 65 00 6C 00  70 00 00 00 00 00 00 00  &.H.e.l.p.......
00000130  00 00 00 00 00 00 00 00  80 08 81 50 0E 00 3A 00  ...........P..:.
00000140  3B 00 0E 00 2F 25 00 00  FF FF 81 00 00 00 00 00  ;.../%..........
00000150  00 00 00 00 00 00 00 00  00 00 02 50 0E 00 30 00  ...........P..0.
00000160  1E 00 08 00 EE 25 00 00  FF FF 82 00 46 00 69 00  .....%......F.i.
00000170  6C 00 65 00 20 00 54 00  79 00 70 00 65 00 00 00  l.e. .T.y.p.e...
00000180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 02 50  ...............P
00000190  54 00 30 00 2C 00 08 00  EF 25 00 00 FF FF 82 00  T.0.,....%......
000001a0  50 00 61 00 72 00 73 00  69 00 6E 00 67 00 20 00  P.a.r.s.i.n.g. .
000001b0  52 00 75 00 6C 00 65 00  73 00 00 00 00 00 00 00  R.u.l.e.s.......
000001c0  00 00 00 00 00 00 00 00  07 00 00 50 06 00 07 00  ...........P....
000001d0  1A 01 71 00 ED 25 00 00  FF FF 80 00 00 00 00 00  ..q..%..........
000001e0  00 00 00 00 00 00 00 00  00 00 02 50 0E 00 11 00  ...........P....
000001f0  3E 00 08 00 EC 25 00 00  FF FF 82 00 53 00 65 00  >....%......S.e.
00000200  6C 00 65 00 63 00 74 00  20 00 52 00 75 00 6C 00  l.e.c.t. .R.u.l.
00000210  65 00 20 00 46 00 6F 00  72 00 20 00 46 00 69 00  e. .F.o.r. .F.i.
00000220  6C 00 65 00 00 00 00 00  00 00 00 00 00 00 00 00  l.e.............
00000230  80 08 81 50 0E 00 1B 00  08 01 0E 00 EB 25 00 00  ...P.........%..
00000240  FF FF 81 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000250  00 00 02 50 19 00 61 00  37 00 08 00 6B 26 00 00  ...P..a.7...k&..
00000260  FF FF 82 00 00 00 00 00                           ........
```

A little about MIPS

- MIPS
  - Short for Microprocessor without Interlocked Pipeline Stages
- Uses RISC architecture (Reduced Instruction Set Computer).
  - Provide a set of simple and fast instructions
  - Compiler translates instructions into 32-bit instructions for instruction memory.
  - Complex instructions are built out of simple ones by the compiler and assembler.

MIPS Registers

- MIPS is a register-to-register (a.k.a. load-store) architecture
  - Source, destination of ALU operations are registers.
  - To operate on memory, need to load it into a register first, then store the result
- MIPS register file provides 32 registers: 0, 1, 2 , … 31
- In assembly we want something more understandable
  - `$t0`, `$t1`
  - `$v0`, `$v1`
  - `$zero`
  - Can still use `$0`, `$1`, `$2`, …

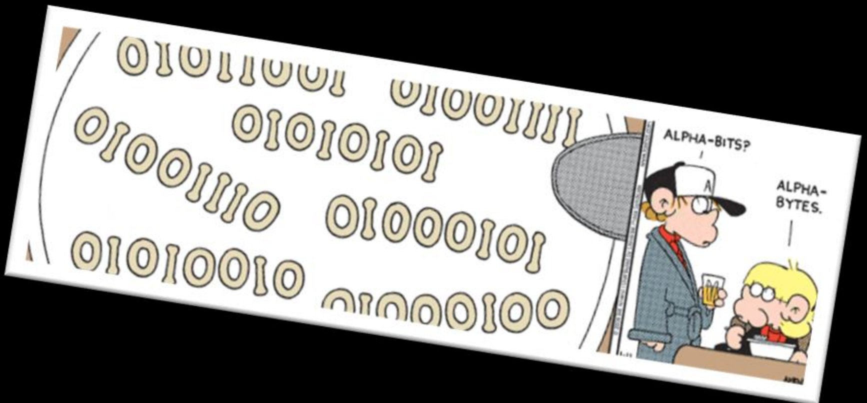
MIPS Registers

- MIPS register file provides 32 registers.
  - Most are used by programs to store values:
    - `$t0 - $t9:` temporaries
    - `$s0 – $s7:` saved temporaries
  - Some have special values:
    - **register 0 (`$zero`) :** always zero (writes to it are discarded)
    - `$at:` reserved for the assembler.
    - `$gp, $sp, $fp, $ra:` memory and function support
    - `$k0, $k1:` reserved for OS kernel
  - Some are used by programs as functions parameters:
    - `$v0, $v1:` return values
    - `$a0–$a3`: function arguments
- Additional special registers (**PC**, **HI**, **LO**) not in register file.
  - **HI** and **LO** are used in multiplication and division
  - These accessed by special instructions.

MIPS Register File Registers

Number	Name	Use
0	`$0, $zero`	Always the constant zero
1	`$at`	reserved for assembler (pseudo instructions)
2 – 3	`$v0 - $v1`	function return values
4 – 7	`$a0 - $a3`	function arguments
8 – 15	`$t0 - $t7`	temporary variables
16 – 23	`$s0 - $s7`	saved temporaries
24 – 25	`$t8 - $t9`	temporary variables
26 – 27	`$k0 - $k1`	reserved for operating system kernel
28	`$gp`	global pointer to data segment
29	`$sp`	stack pointer to top of stack
30	`$fp`	frame pointer to function frame start
31	`$ra`	return address from function

MIPS Memory and Instructions



- All memory is addressed in bytes.
- Instruction addresses too!
  - Starting from the instruction at address 0.
- All instructions are 32 bits (4 bytes) long
- Therefore:
  all instruction addresses are divisible by 4.
- Also, all registers are 4 bytes long (one word)

Recall: MIPS instruction types

- R-type:

opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

- I-type:

opcode	rs	rt	immediate
6	5	5	16

- J-type:

opcode	address
6	26

Common Asm Instructions

Instruction Type	Examples	Usage	Integer Frequency	Floating point Frequency
Arithmetic	`add, sub, addi`	Operations in assignment statement s	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	References to data structures, such as arrays	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	operations in assignment statement s	12%	4%
Conditional branch	`beq, bne, slt, slti, sltiu`	If statements and loops	34%	8%
Jump	`j, jr, jal`	Procedure calls, returns, and case/switch statements	2%	0%

Source: ***Computer Organization And Design: The Hardware/Software Interface*, 5^{th} Edition, Patterson & Hennessy, 2014, p163**

Types of Asm Instructions

Arithmetic	`add, mult, …`
Logical	`and, or, …`
Bit shifting	`sll, sra, …`
Data movement	`mflo,mfhi, …`
Branch	`beq, bgtz, …`
Jump	`j, jr, …`
Comparison	`slt, sltu, …`
Memory	`lw, sw, …`

Microarchitecture vs. ISA

- Assembly has 8 types of instructions, yet we know there are 3 types of MIPS instructions?
- Instruction Set Architecture (ISA)
  - The set of instructions supported by a processor.
  - Registers, address and data formats, execution model, etc.
  - A "contract" between processor and programmer.
- Microarchitecture
  - The implementation of the ISA on a processor.
  - Cache, pipeline, branch prediction, datapath…
- Computer architecture: combination of both.

ALU Instructions

Source: [Computer History Museum]

Arithmetic instructions

Instruction	Opcode/Function	Syntax	Operation
add	100000	$d, $s, $t	$d = $s + $t
addu	100001	$d, $s, $t	$d = $s + $t
addi	001000	$t, $s, i	$t = $s + SE(i)
addiu	001001	$t, $s, i	$t = $s + SE(i)
div	011010	$s, $t	lo = $s / $t; hi = $s % $t
divu	011011	$s, $t	lo = $s / $t; hi = $s % $t
mult	011000	$s, $t	hi:lo = $s * $t
multu	011001	$s, $t	hi:lo = $s * $t
sub	100010	$d, $s, $t	$d = $s - $t
subu	100011	$d, $s, $t	$d = $s - $t

Notes: “hi” and “lo” refer to the HI and LO registers (see register slide). “SE” = “sign extend”.

R-type vs I-type arithmetic

R-Type

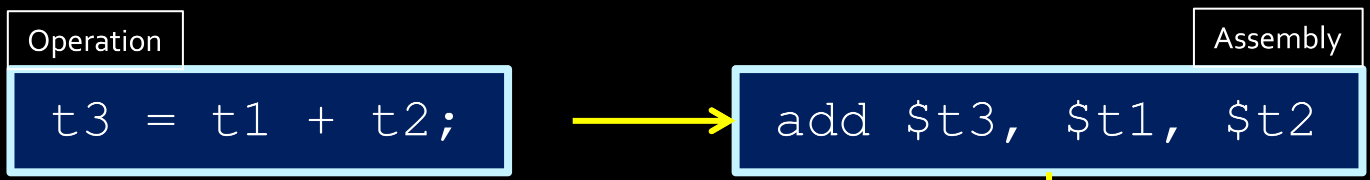
- `add, addu`
- `div, divu`
- `mult, multu`
- `sub, subu`

I-Type

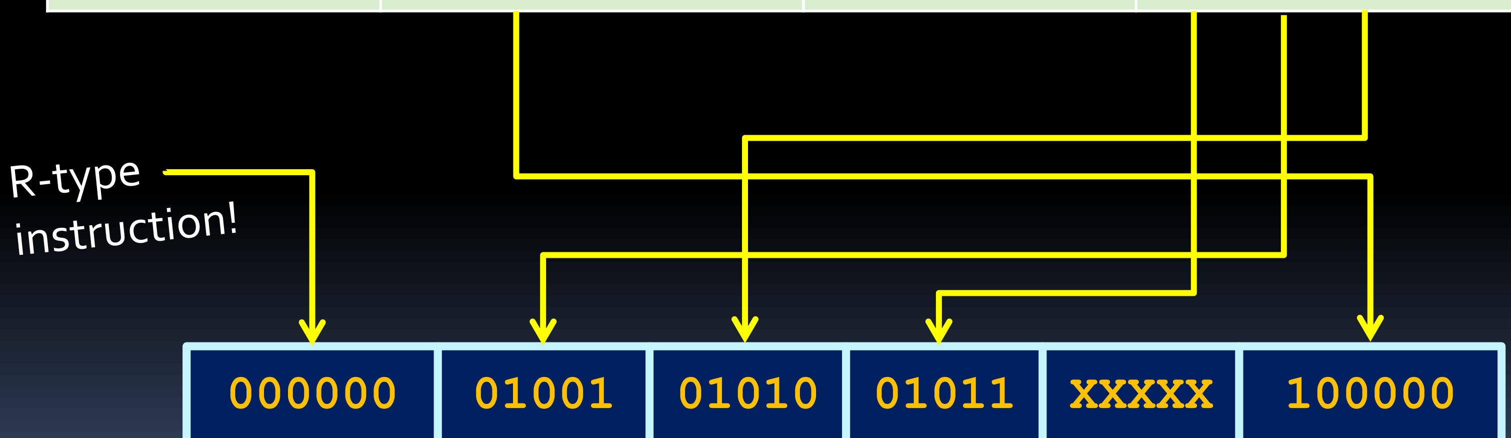
- `addi`
- `addiu`

- In general, most instructions are R-type (meaning all operands are registers) and some are I-type (meaning they use an immediate/constant value in their operation).
- Can you recognize which of the following are R-type and I-type instructions? (Hint: “i” for “immediate”)

Assembly → Machine Code



Instruction	Opcode/Function	Syntax	Operation
add	100000	$d, $s, $t	$d = $s + $t



Although we specify "don't care" bits as X values, the assembler generally assigns some value (like 0).

Unsigned Instructions

- What is the difference between `add` or `addu`?
  - Both do exactly same thing! Add numbers.
- "`u`" stands for "`unsigned`"
  - Causes a "trap" (a.k.a exception) if there is overflow
  - Stops execution of current code.
  - addu ignores this overflow
- `mult` and `multu` are not the same!
  - Slight difference in operation. Use the right one!
  - Neither check for overflow.

Logical instructions

Instruction	Opcode/Function	Syntax	Operation
and	100100	$d, $s, $t	$d = $s & $t
andi	001100	$t, $s, i	$t = $s & ZE(i)
nor	100111	$d, $s, $t	$d = ~($s | $t)
or	100101	$d, $s, $t	$d = $s | $t
ori	001101	$t, $s, i	$t = $s | ZE(i)
xor	100110	$d, $s, $t	$d = $s ^ $t
xori	001110	$t, $s, i	$t = $s ^ ZE(i)

Note: ZE = zero extend (pad upper bits with 0 value).

Assembly → Machine Code II

Operation

`t2 = t1 & 42;`

Assembly

`andi $t2, $t1, 42`

Instruction	Opcode/Function	Syntax	Operation
`andi`	001100	\$t, \$s, i	\$t = \$s & ZE(i)

I-type instruction!

001100	01001	01010	0000000000101010

Shift instructions

Instruction	Opcode/Function	Syntax	Operation
`sll`	000000	$d, $t, a	$d = $t << a
`sllv`	000100	$d, $t, $s	$d = $t << $s
`sra`	000011	$d, $t, a	$d = $t >> a
`srav`	000111	$d, $t, $s	$d = $t >> $s
`srl`	000010	$d, $t, a	$d = $t >>> a
`srlv`	000110	$d, $t, $s	$d = $t >>> $s

Note: `srl` = "shift right logical"
`sra` = "shift right arithmetic".
The "`v`" denotes a variable number of bits, specified by `$s`.
`a` is shift amount, and is stored in `shamt` when encoding the R-type machine code instructions.

Data movement instructions

Instruction	Opcode/Function	Syntax	Operation
mfhi	010000	$d	$d = hi
mflo	010010	$d	$d = lo
mthi	010001	$s	hi = $s
mtlo	010011	$s	lo = $s

- These are instructions for operating on the HI and LO registers described earlier (for multiplication and division)

lui – load upper immediate

Instruction	Opcode/Function	Syntax	Operation
lui	001111	$t, i	$t = i << 16

- Load 16-bit immediate into upper half of the register.
- The lower 16 bits of the register are set to zero.

iiiiiiiiiiiiiiii0000000000000000

ALU instructions in RISC

- Most ALU instructions are R-type instructions.
  - The six-digit codes in the tables are therefore the function codes (opcodes are `000000`).
  - Exceptions are the I-type instructions (`addi`, `andi`, `ori`, etc.)
- Not all R-type instructions have an I-type equivalent.
  - RISC principle dictate that an operation doesn't need an instruction if it can be performed through multiple existing operations.
  - Example `addi` + `div` → `divi`

Pseudoinstructions

- Move data from $t4 to $t5?
  - `move $t5,$t4` →

```
add $t5,$t4,$zero
```

- Multiply and store in $s3?
  - `mul $s1, $t4, $t5` →

```
mult $t4,$t5
mflo $s1
```

- Load a 32-bit immediate?
  - `li $s0,0x1234ABCD` →

```
lui $s0,0x1234
ori $s0,0xABCD
```

Pseudoinstructions

- Pseudo instructions look like assembly instructions…
- …but don't have a dedicated machine code instruction.
- Provided by the assembler
  - Mapping ASM to machine code is more like a many-to-one mapping…
- If a temporary register is needed, use $at

Making an assembly program

- Assembly language programs typically have structure similar to simple Python or C programs:
  - They set aside registers to store data.
  - They have sections of instructions that manipulate this data.
- It is always good to decide at the beginning which registers will be used for what purpose!
  - More on this later ☺