Week 7, part E: executing instructions

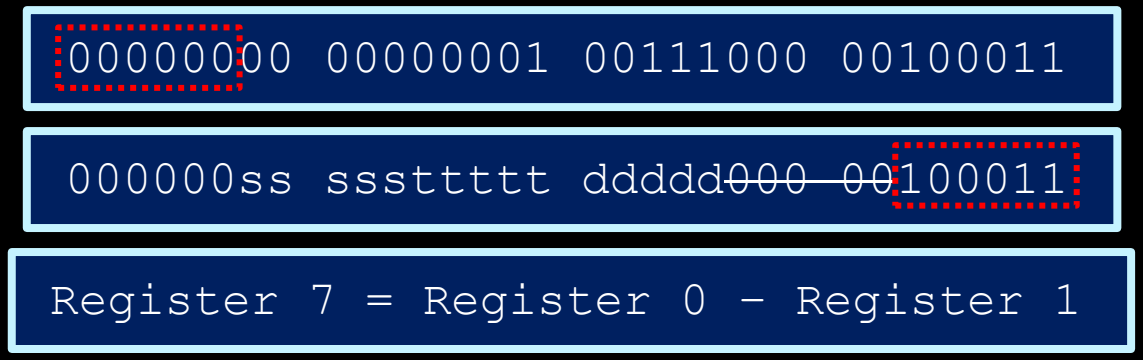
Understanding instructions

- How do we decode this binary instruction?

```
00000000 00000001 00111000 00100011
```

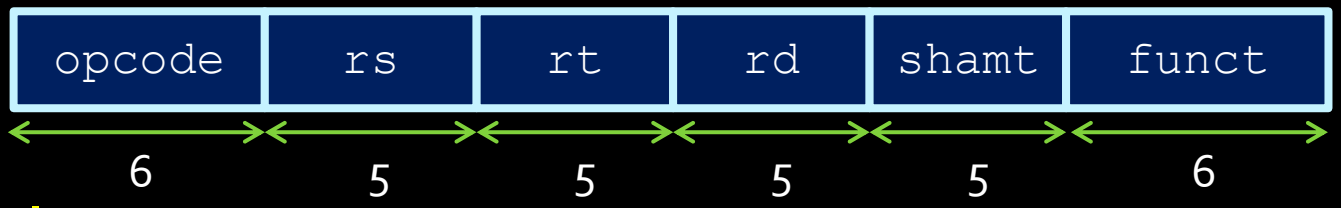
Instruction decoding

- Instructions are comprised of different parts, and contain all the information needed to:
  1. Decode the instruction.
  2. Execute the operation.
- Example: unsigned subtraction



MIPS instruction types

- R-type:



- I-type:

- J-type:

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

Opcodes

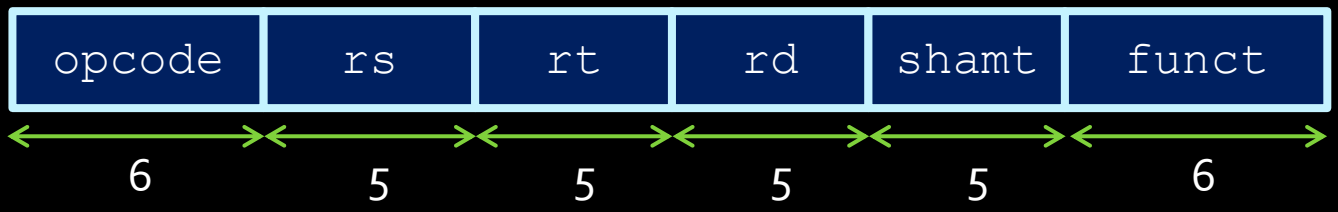
- Same table as before.
- Colors indicate: R-type, I-type, J-type
- R-type opcode always 000000 so instead we list the funct field .

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

Decoding Instructions

1. Look at opcode (first 6 bits)

2. If oooooo
   → R-type

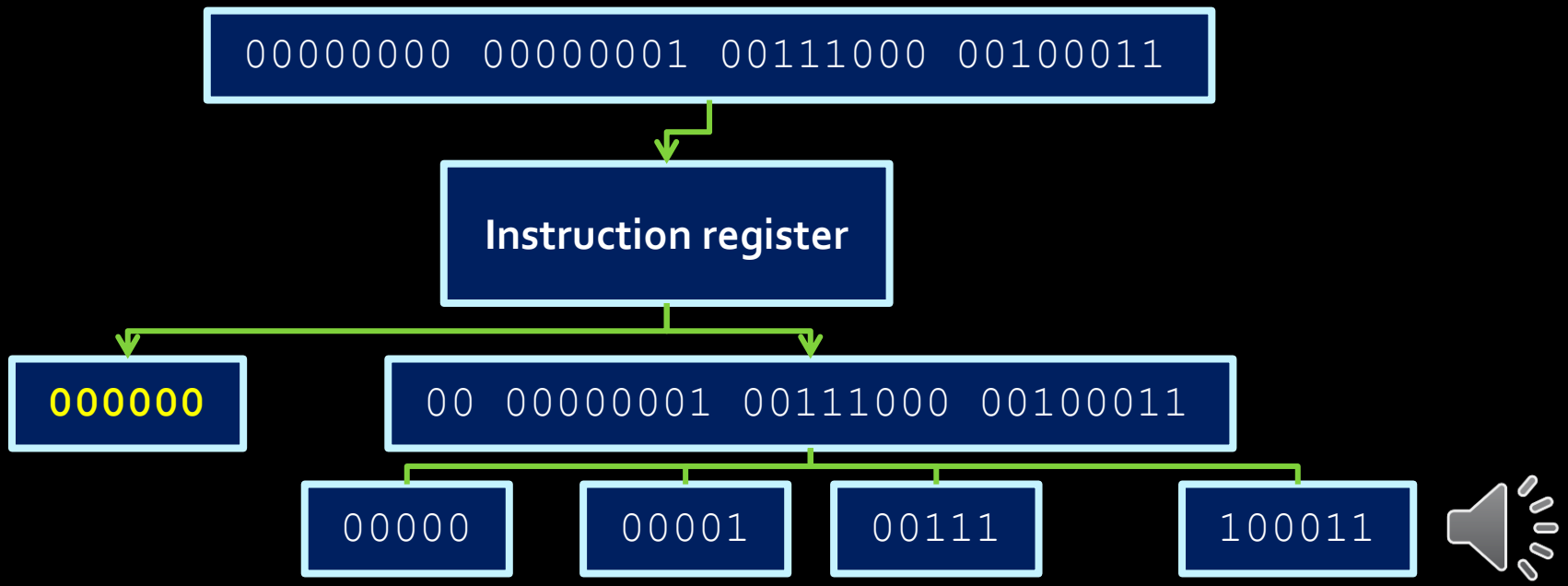


3. Otherwise lookup opcode in table:
   → I-type

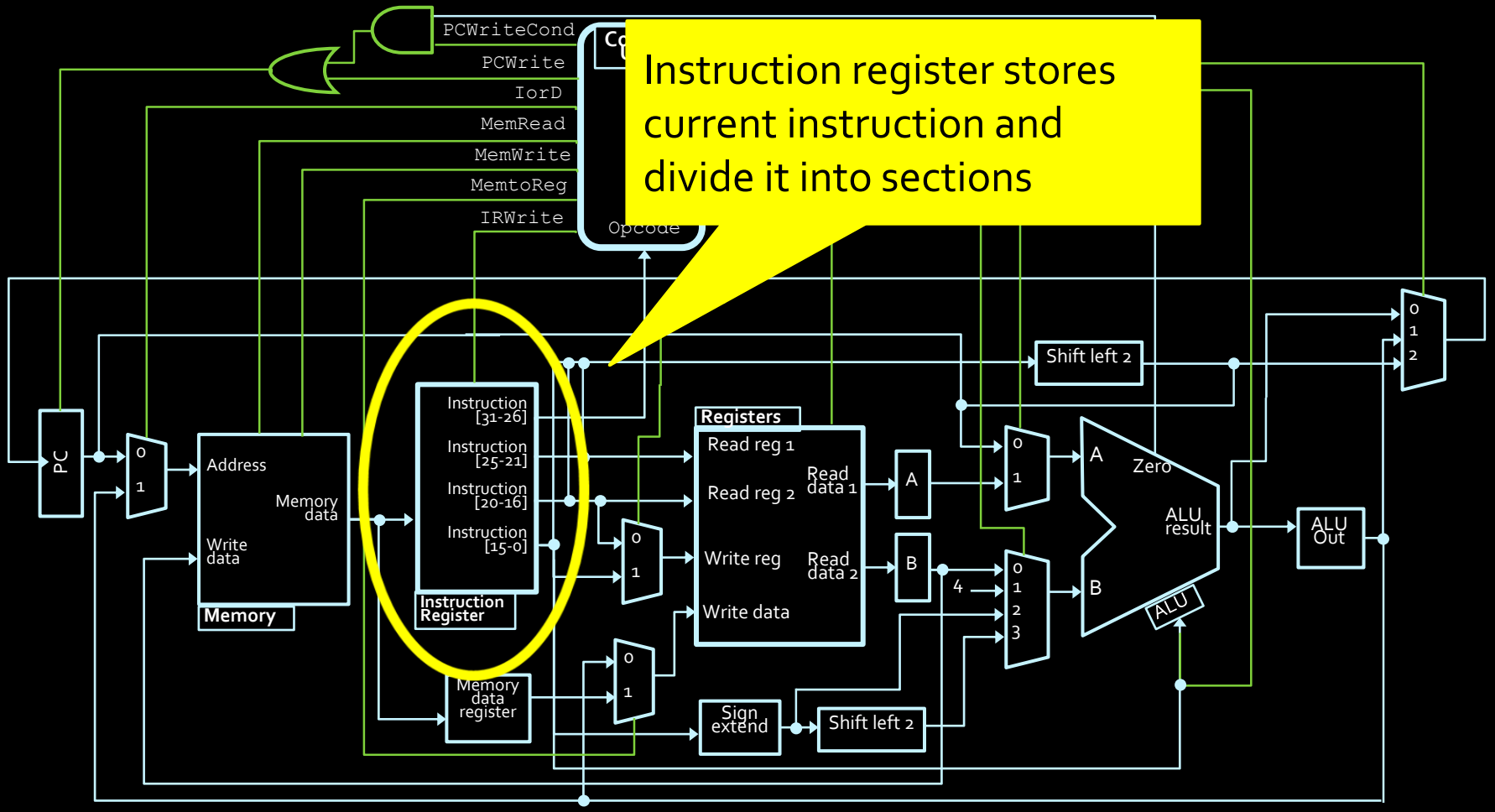
   → J-type

Instruction registers

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



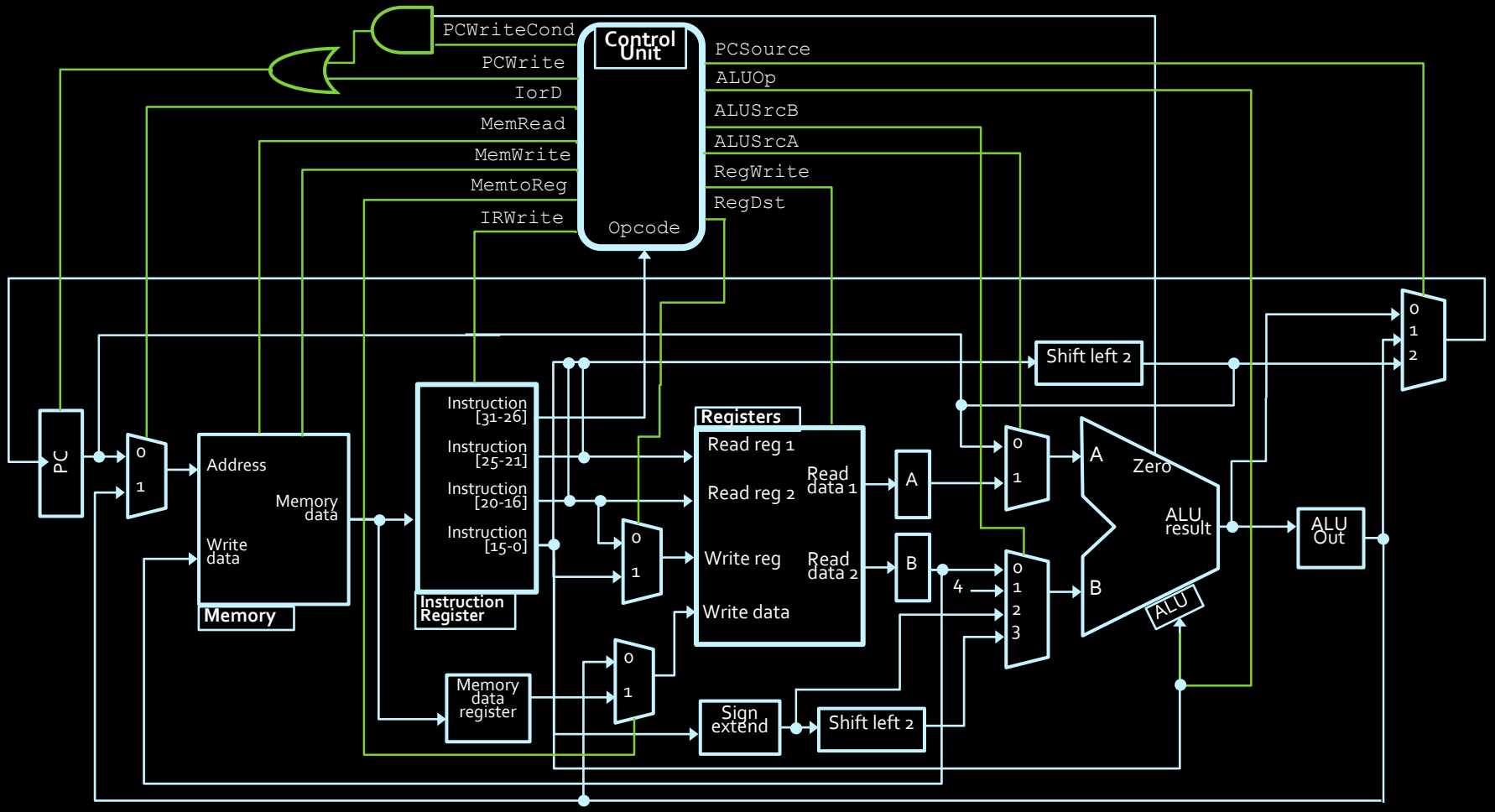
The Instruction Register



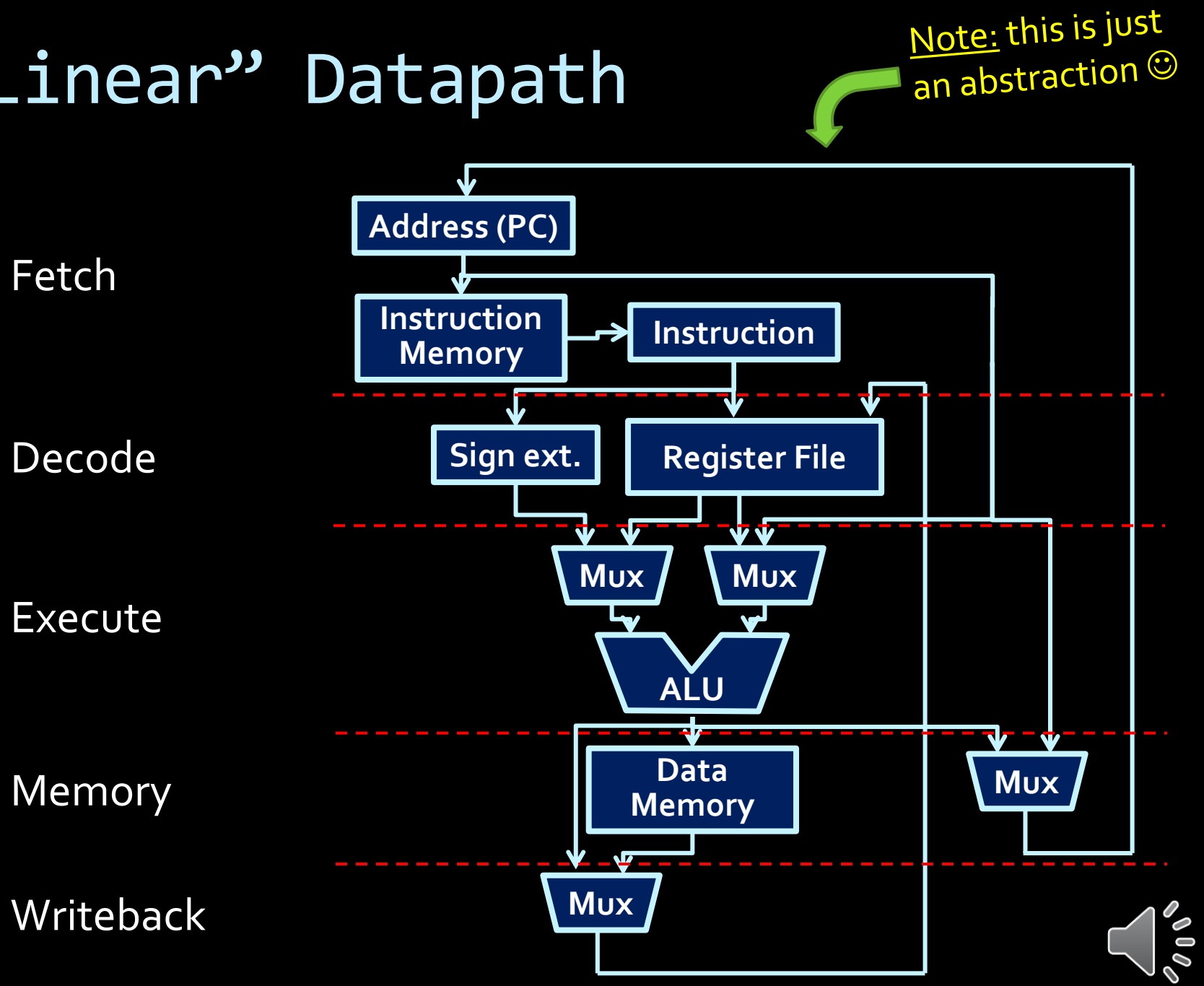
Abstract Diagram of Datapath

- Let's simplify the MIPS datapath.
- Divide each execution to 5 stages:
  1. Fetch instruction
  2. Decode and register file fetch
  3. Execute computation on data or address
  4. Memory access
  5. Write back to registers
- Also: pretend we have separate memories: one for instruction and one for data.

The Processor Datapath



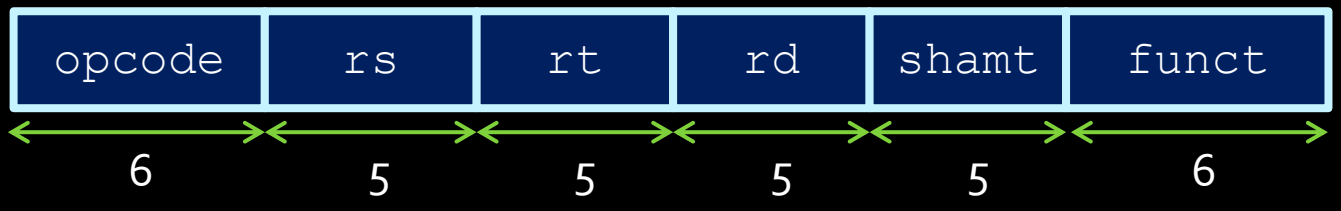
"Linear" Datapath



How Instructions Work

- We'll now discuss the 3 types of instructions:
  - R-type
  - I-type
  - J-type
- And we'll now how they use the datapath.

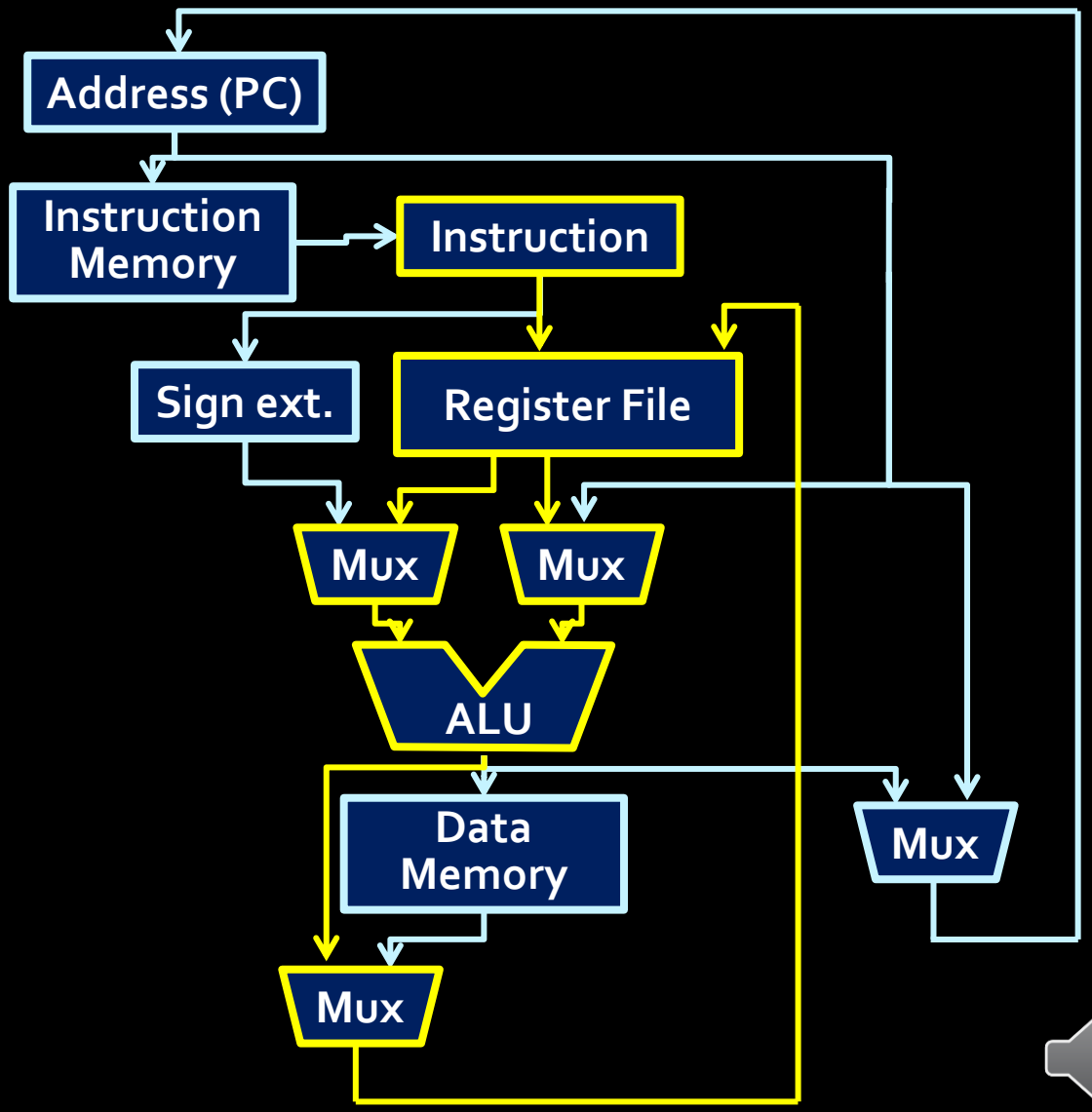
R-type instructions



- Short for "register-type" instructions.
  - Because they operate on the registers, naturally.
- The `opcode` for all R-type instructions is `000000`.
- Fields for specifying:
  - Two source registers (`rs` and `rt`)
  - One destination register (`rd`).
  - Shift amount for shifting instructions (`shamt`).
- If we don't need a field, fill it with `0` bits.
- The `funct` 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.
- `rd` is also sent to the register file, to specify the location of the result.



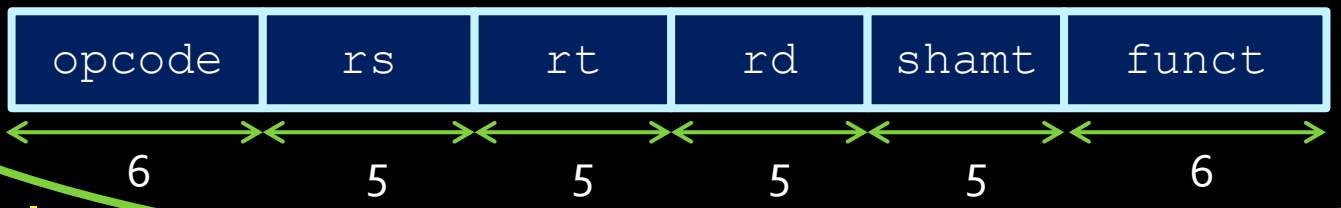
Example

- Decode this instruction:

- We look at opcode, it is oooooo → R-type

MIPS instruction types

- R-type:



- I-type:

- J-type:

Example

opcode	rs	rt	rd	shamt	funct
000000	00 110	10001	00101	000 00	100110

- We look at opcode, it is oooooo → R-type
- Now look at funct → 100110

100110 is XOR →

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

Example

```
000000 00 11010001 00101 000 00100110
```

- We look at opcode, it is oooooo → 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 6 and 17 and store it in register 5 (`xor $5, $17, $6`)

I-type instructions

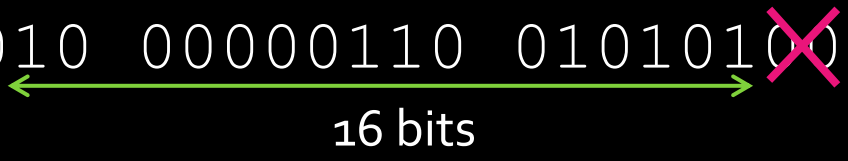
- I-type instructions have a 16-bit immediate field.
- This field is a constant value, which is used for:
  - Arithmetic or other operations:
    - e.g., to compute `$rt = $rs + <immediate>`
  - a displacement (offset) for memory address.
    - e.g, write to memory address `$rs + <immediate>`
  - a branch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

A Trick for 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

Example

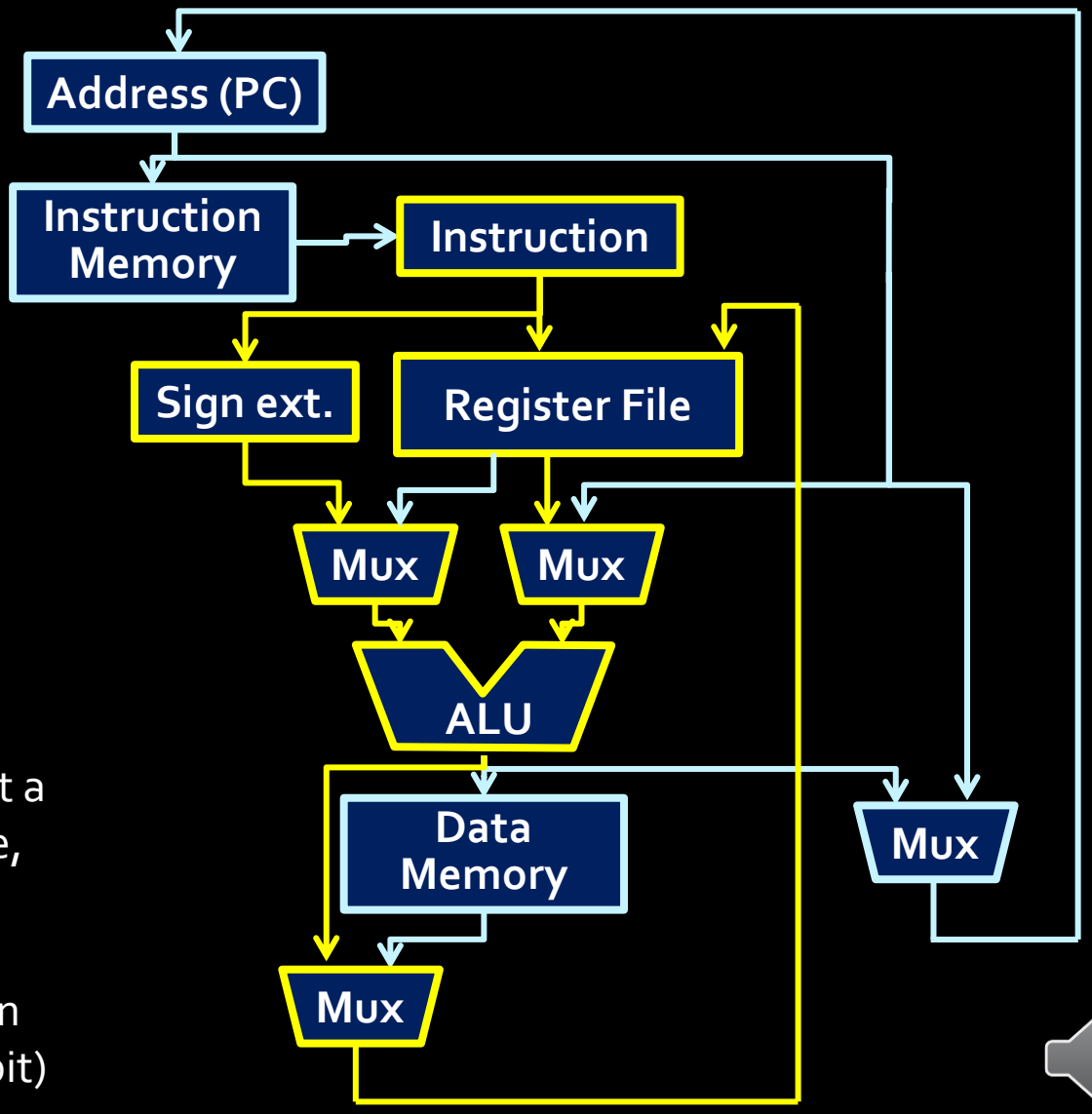
- We are currently at address `0x0300B004`
- We want to branch to target `0x0302B658`
- We cannot store target in 16 bits
- Compute the difference target - current:
  `0x0302B658 – 0x0302B004 = 0x20654`
  - Bin: `00000000 00000010 00000110 01010100`
  - Discard lower 2 bits
  - Take 16 bits that were left



- Immediate field: `1000000110010101`

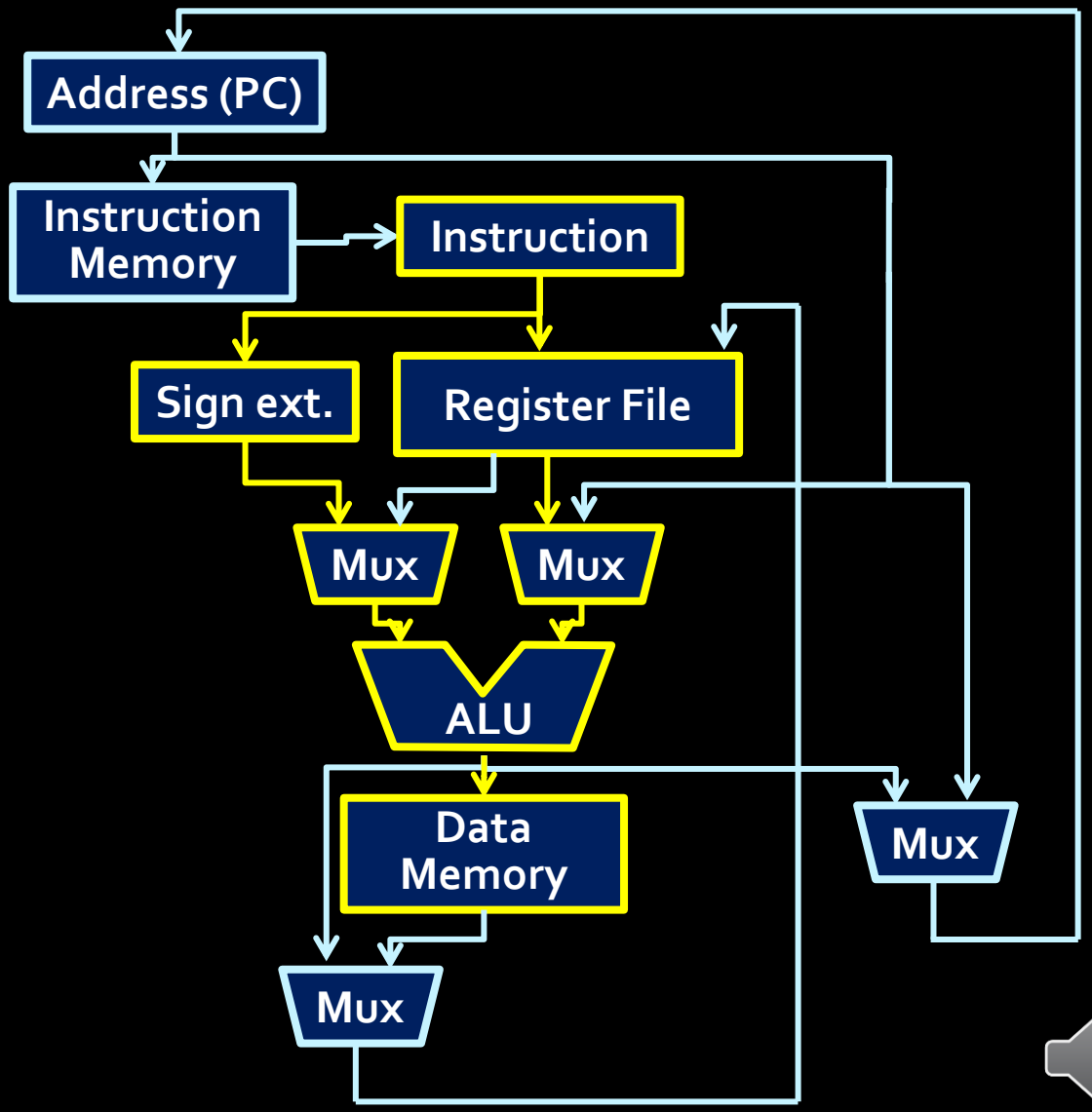
I-type instruction datapath

- 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)



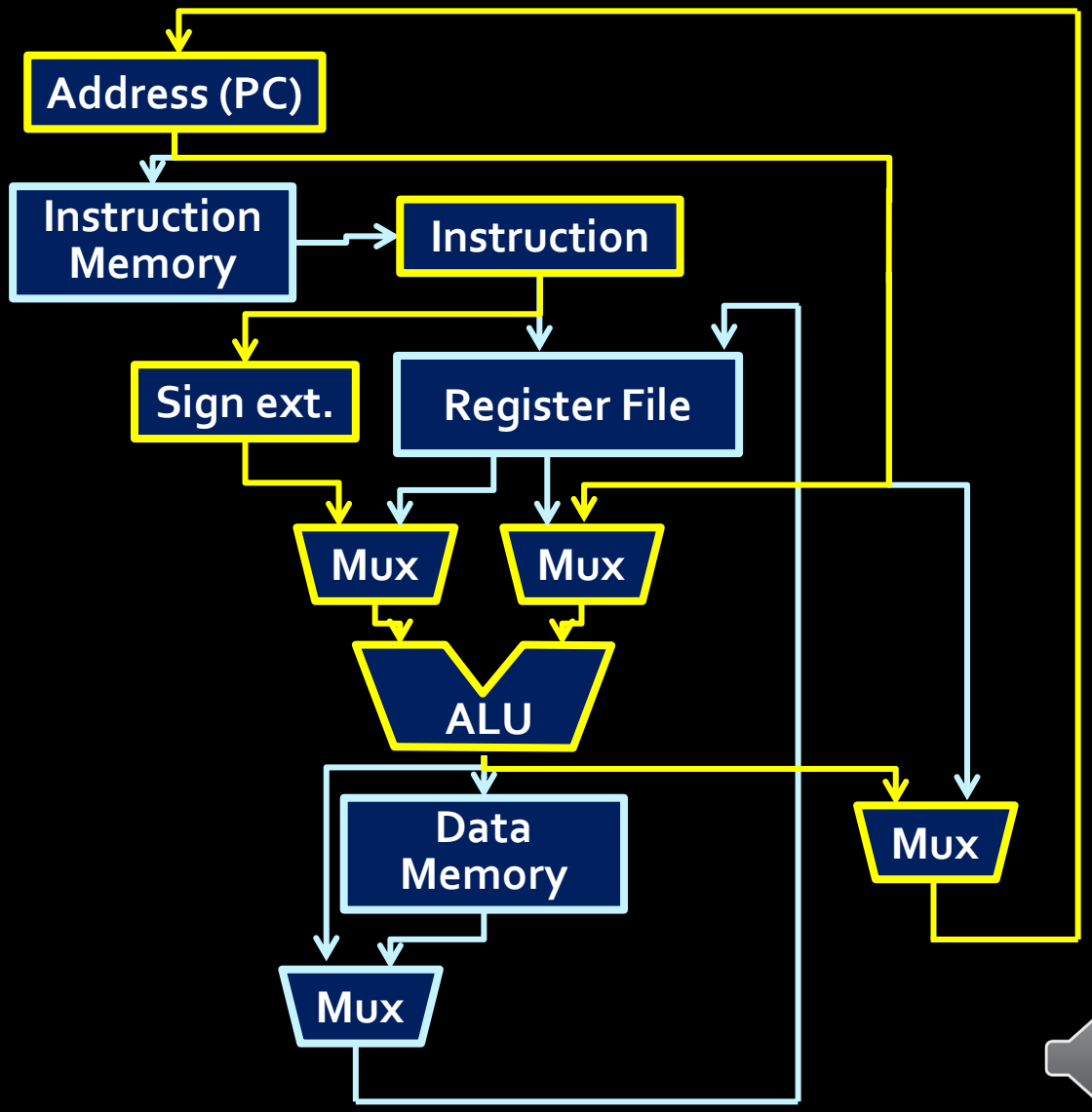
I-type instruction datapath

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



I-type instruction datapath

- Example #3: Branch instructions.
  - Output (new address) is written to PC
  - It will be used to fetch the next instruction.



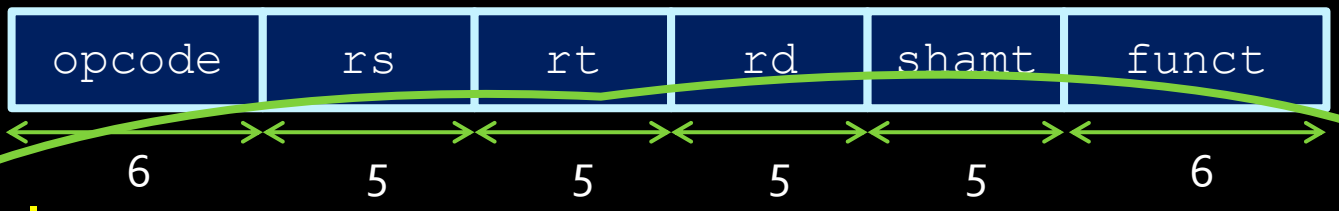
Example Decoding

```
00100000 11010001 00000000 00100110
```

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:



- I-type:

- J-type:

Example

opcode	rs	rt	immediate
001000	00 110	10001	00000000 00100110

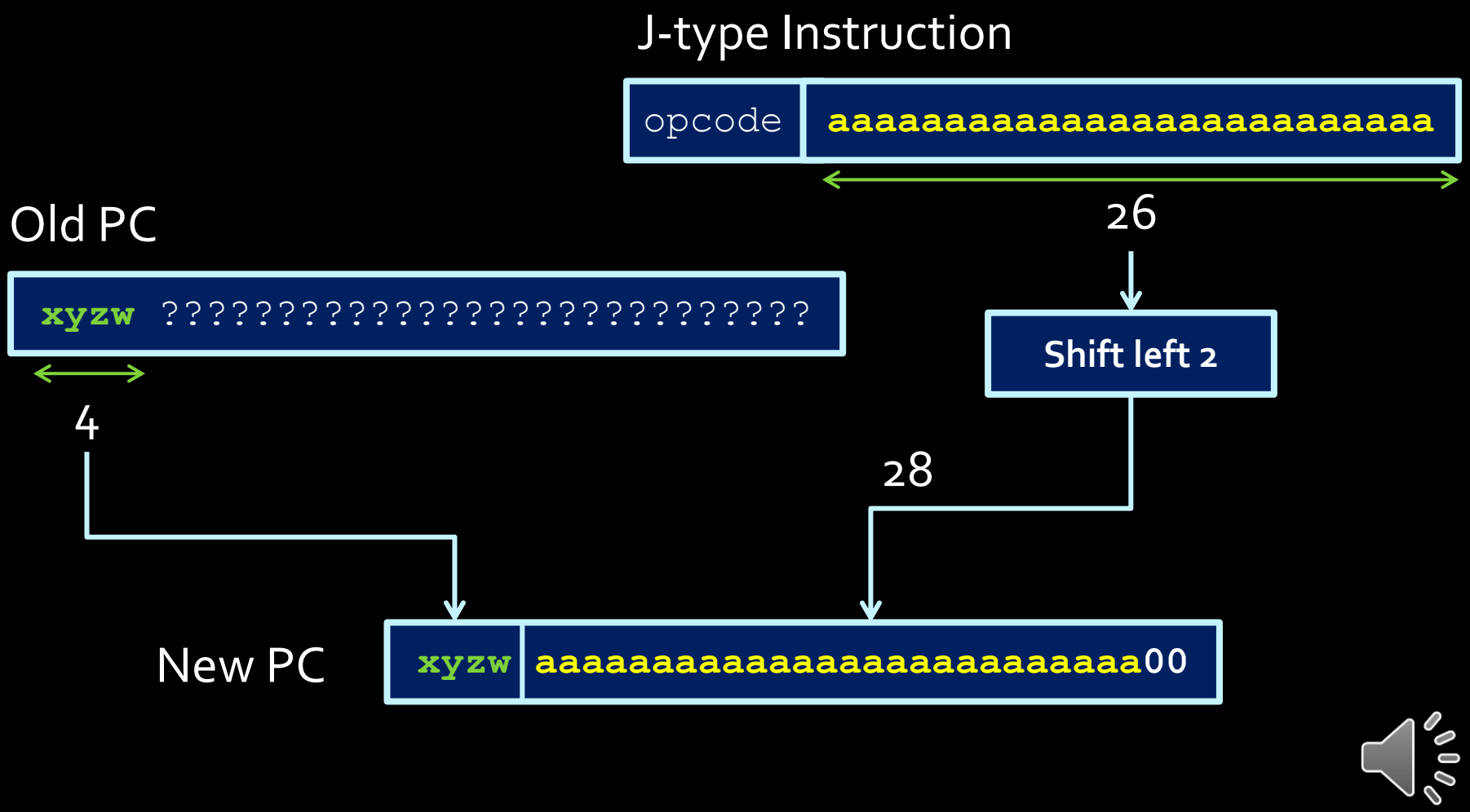
- Opcode 001000 → I-type → addi
- **rs = 6**
- **rt = 17**
- **Immediate = 38**

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

J-type instructions

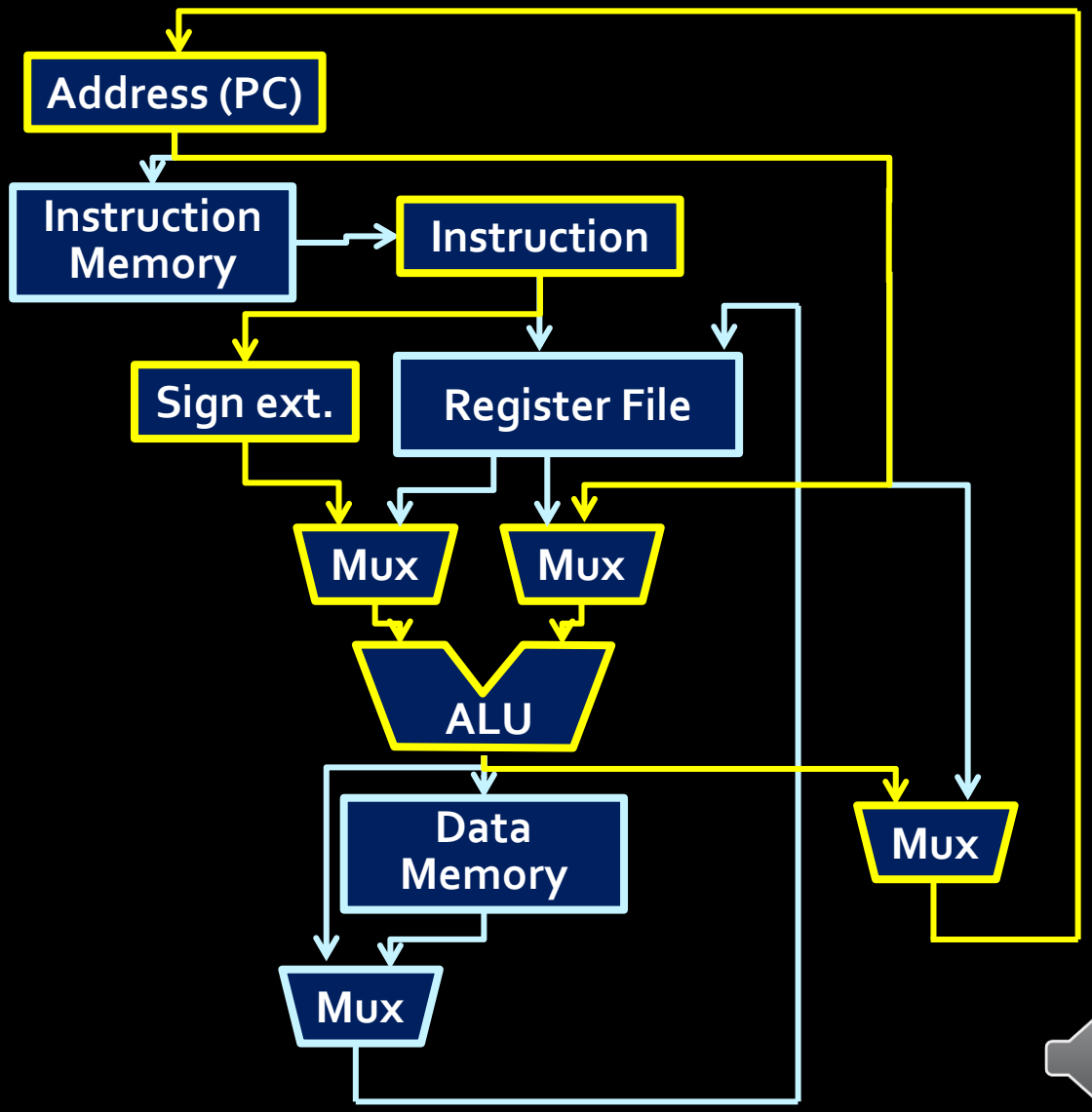
- Only two J-type instructions:
  - jump (`j`)
  - jump and link (`jal`)
- These instructions use the 26-bit coded address field to specify the target of the jump.
- But 32 bits are needed to encode the target address.
  - The bits at positions `1` and `0` are always `0` (word alignment) .
  - Take bits from positions `27` to `2` in the target address and use them as the 26 bits provided in the instruction.
  - This gives us 28 bits. The remaining highest 4 bits cannot be encoded.
  - Use the highest 4 bits of the current PC as the highest 4 bits of the target address.

Jump Instructions



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 on top of the old PC value.



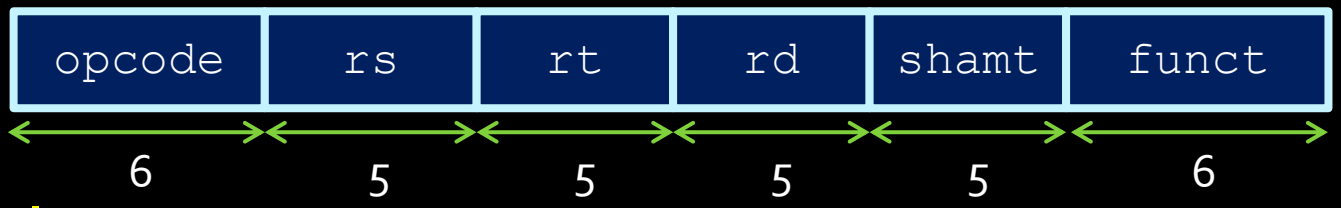
Examples

```
00001010 11010001 00000000 00100110
```

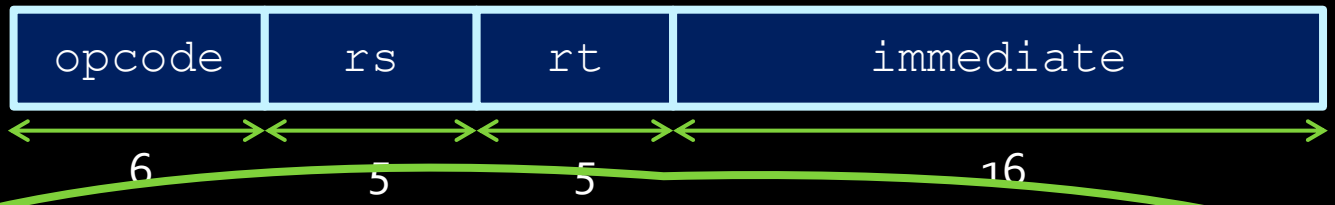
Instruction	Op/Func	Instruction	Op/Func
add	100000	srav	000111
addu	100001	srl	000010
addi	001000	srlv	000110
addiu	001001	beq	000100
div	011010	bgtz	000111
divu	011011	blez	000110
mult	011000	bne	000101
multu	011001	j	000010
sub	100010	jal	000011
subu	100011	jalr	001001
and	100100	jr	001000
andi	001100	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:



- 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`)

Takeaway

Different instructions have different flows in the datapath.

In other words, if we can control the paths of flow, then we can control what instruction to execute.

Almost There

- We know what the CPU does:
  - Fetch instructions from PC
  - Decode and execute them
  - Move to the next instruction
- We now need to understand the control signals to make all this happen.
- Move to next part!