# A Processor

**Hakim Weatherspoon**
**CS 3410, Spring 2012**
Computer Science
Cornell University

See: P&H Chapter 2.16-20, 4.1-4

# Administrivia

***Required***: partner for group project

Project1 (PA1) and Homework2 (HW2) are both out

PA1 Design Doc and HW2 due in one week, start early

Work alone on HW2, but in group for PA1

Save your work!

- ***Save often***. Verify file is non-zero. Periodically save to Dropbox, email.
- Beware of MacOSX 10.5 (leopard) and 10.6 (snow-leopard)

Use your resources

- Lab Section, Piazza.com, Office Hours, Homework Help Session,
- Class notes, book, Sections, CSUGLab

# Administrivia

Check online syllabus/schedule

- http://www.cs.cornell.edu/Courses/CS3410/2012sp/schedule.html

Slides and Reading for lectures

Office Hours

Homework and Programming Assignments

Prelims (in evenings):

- Tuesday, February 28th
- Thursday, March 29th
- Thursday, April 26th

Schedule is subject to change

# Collaboration, Late, Re-grading Policies

## "Black Board" Collaboration Policy

- Can discuss approach together on a "black board"
- Leave and write up solution independently
- Do not copy solutions

## Late Policy

- Each person has a total of ***four*** "slip days"
- Max of ***two*** slip days for any individual assignment
- Slip days deducted first for *any* late assignment, cannot selectively apply slip days
- For projects, slip days are deducted from all partners
- 20% deducted per day late after slip days are exhausted

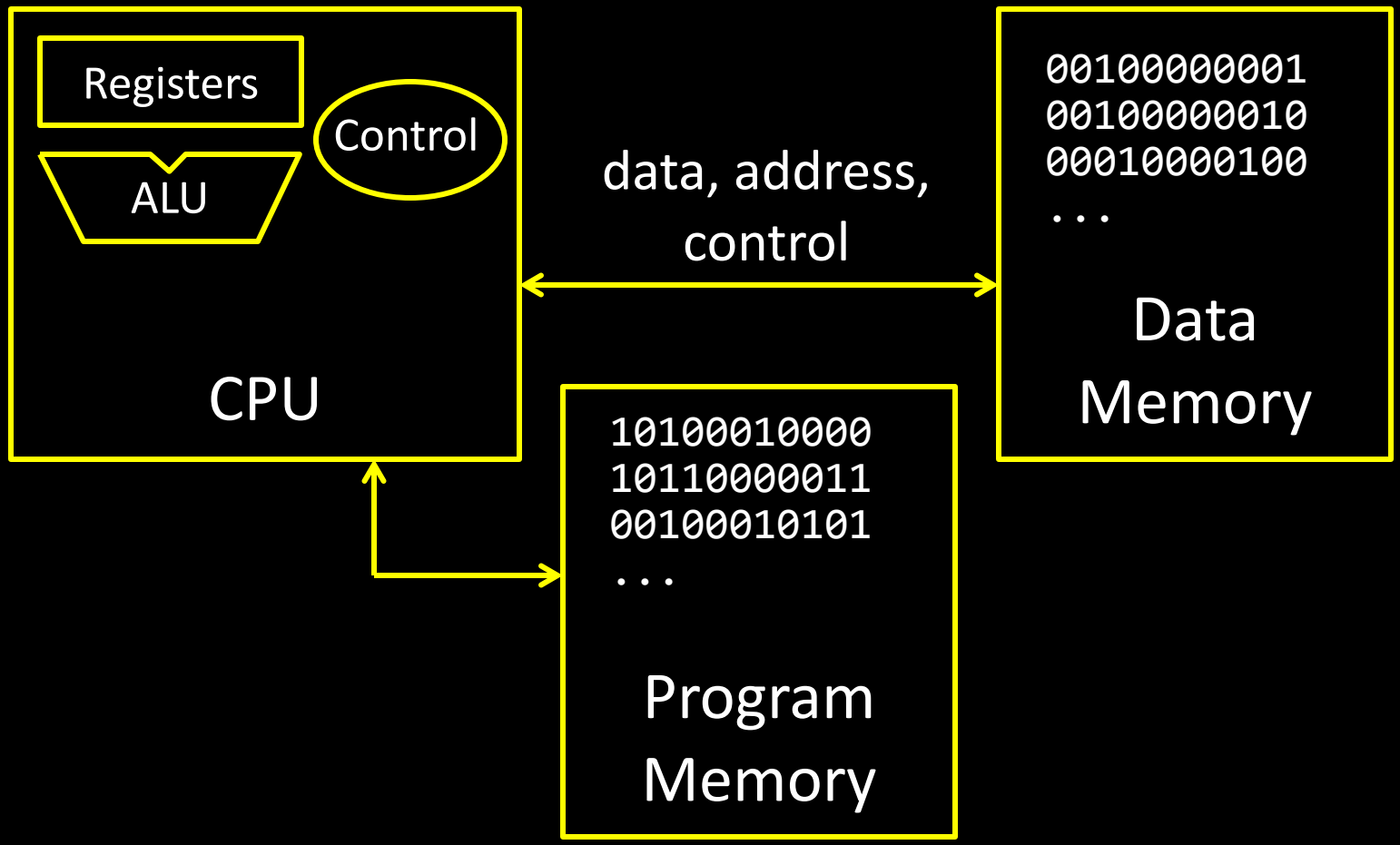
## Regrade policy

- Submit written request to lead TA, and lead TA will pick a different grader
- Submit another written request, lead TA will regrade directly
- Submit yet another written request for professor to regrade.

# Basic Computer System

Let's build a MIPS CPU

- …but using (modified) Harvard architecture



# Instructions

```
for (i = 0; i < 10; i++)
        printf("go cucs");
```

```
main:  addi r2, r0, 10
       addi r1, r0, 0
loop:  slt r3, r1, r2
       ...
```

```
00100000000000100000000000001010
00100000000000100000000000000000
00000000001000100001100000101010
```

## High Level Language

- C, Java, Python, Ruby, …
- Loops, control flow, variables

## Assembly Language

- No symbols (except labels)
- One operation per statement

## Machine Langauge

- Binary-encoded assembly
- Labels become addresses

# Instruction Types

Arithmetic

- add, subtract, shift left, shift right, multiply, divide

Memory

- load value from memory to a register
- store value to memory from a register

Control flow

- unconditional jumps
- conditional jumps (branches)
- jump and link (subroutine call)

Many other instructions are possible

- vector add/sub/mul/div, string operations
- manipulate coprocessor
- I/O

# Complexity

## MIPS = Reduced Instruction Set Computer (RISC)

- ≈ 200 instructions, 32 bits each, 3 formats
  - mostly orthogonal
- all operands in registers
  - almost all are 32 bits each, can be used interchangeably
- ≈ 1 addressing mode: Mem[reg + imm]

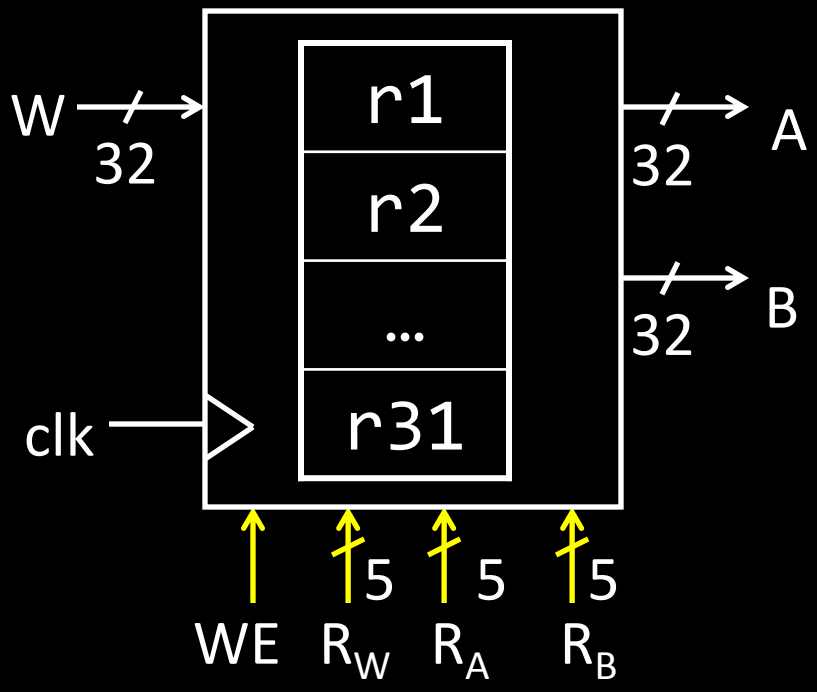
## x86 = Complex Instruction Set Computer (CISC)

- > 1000 instructions, 1 to 15 bytes each
- operands in special registers, general purpose registers, memory, on stack, …
  - can be 1, 2, 4, 8 bytes, signed or unsigned
- 10s of addressing modes
  - e.g. Mem[segment + reg + reg*scale + offset]

# MIPS Register file

## MIPS register file

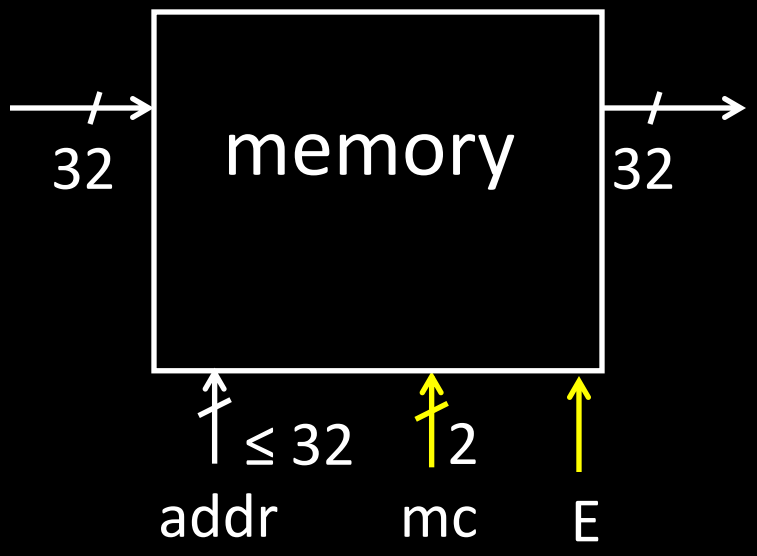
- 32 registers, 32-bits each (with r0 wired to zero)
- Write port indexed via $R_W$
  - Writes occur on falling edge but only if WE is high
- Read ports indexed via $R_A$, $R_B$



# MIPS Memory

## MIPS Memory

- Up to 32-bit address
- 32-bit data
  (but byte addressed)
- Enable + 2 bit memory control
  - 00: read word (4 byte aligned)
  - 01: write byte
  - 10: write halfword  (2 byte aligned)
  - 11: write word (4 byte aligned)



# Instruction Usage

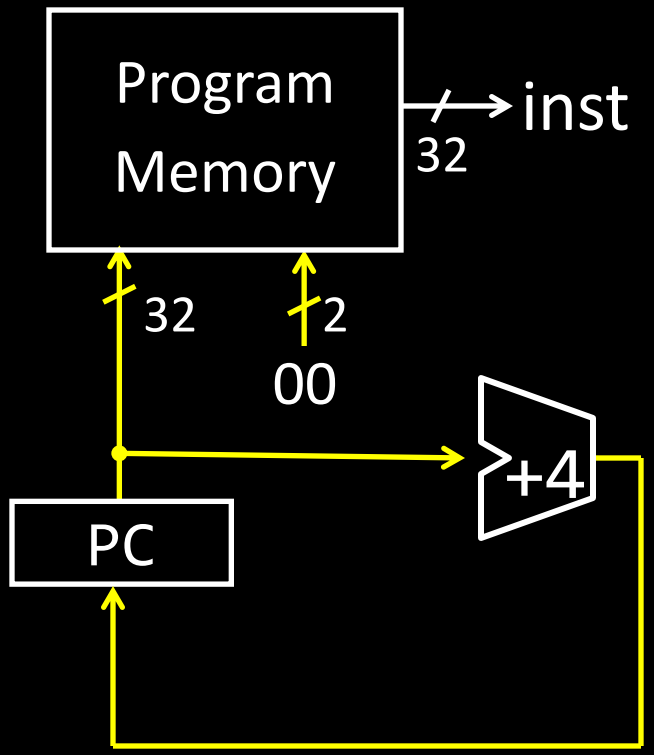
Basic CPU execution loop

1. fetch one instruction
2. increment PC
3. decode
4. execute

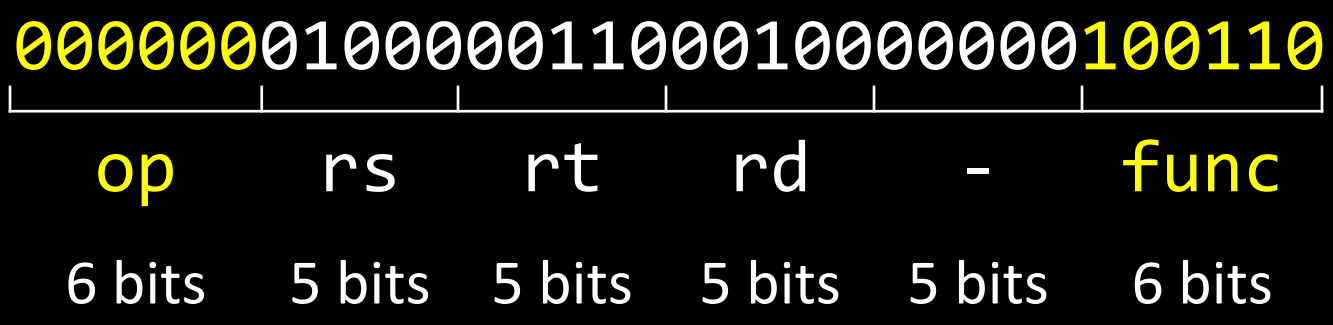
# Instruction Fetch

## Instruction Fetch Circuit

- Fetch instruction from memory
- Calculate address of next instruction
- Repeat

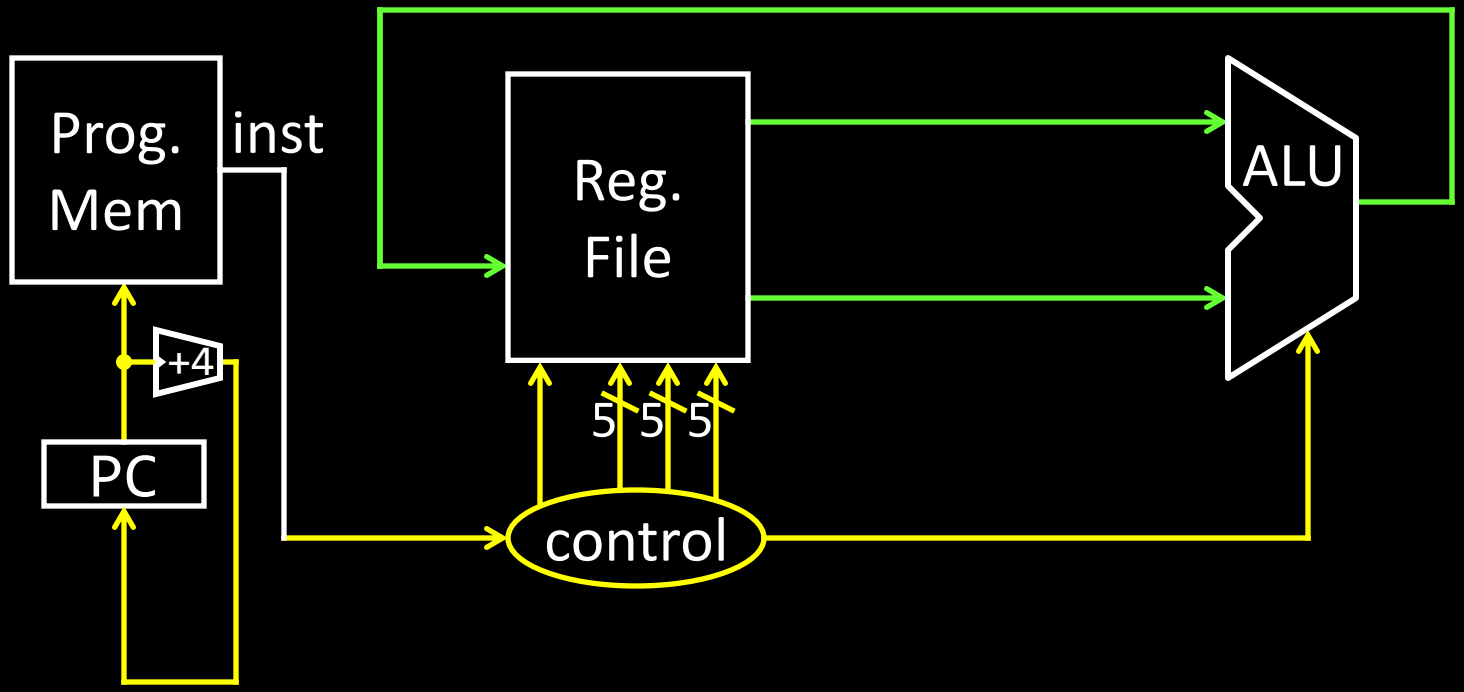


# Arithmetic Instructions




| op | func | mnemonic | description |
|---|---|---|---|
| 0x0 | 0x21 | ADDU rd, rs, rt | R[rd] = R[rs] + R[rt] |
| 0x0 | 0x23 | SUBU rd, rs, rt | R[rd] = R[rs] – R[rt] |
| 0x0 | 0x25 | OR rd, rs, rt | R[rd] = R[rs] \| R[rt] |
| 0x0 | 0x26 | XOR rd, rs, rt | R[rd] = R[rs] ⊕ R[rt] |
| 0x0 | 0x27 | NOR rd, rs rt | R[rd] = ~ ( R[rs] \| R[rt] ) |

# Arithmetic and Logic



# Example Programs

r4 = (r1 + r2) | r3

r8 = 4*r3 + r4 – 1

r9 = 9

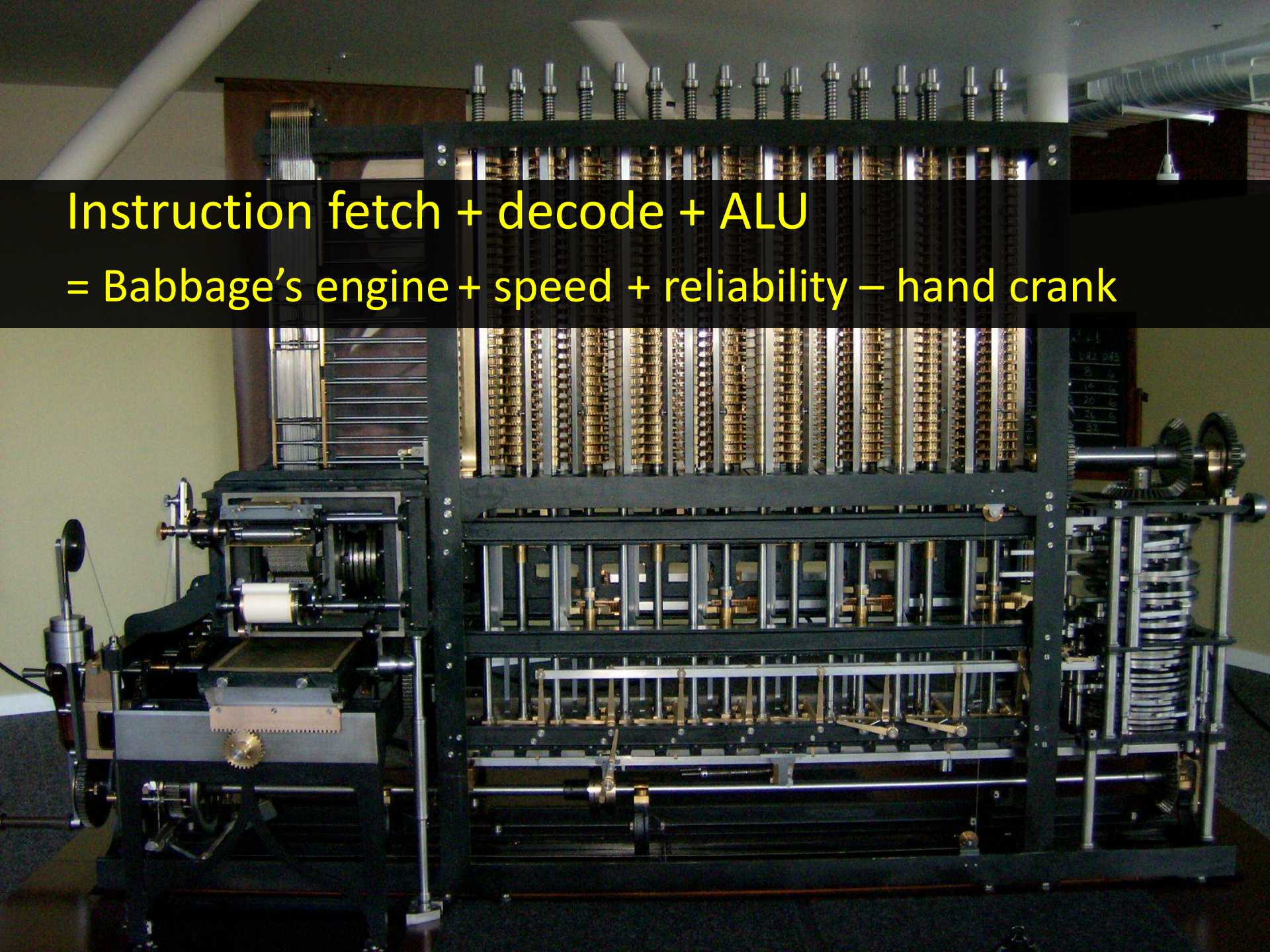
ADDU rd, rs, rt

SUBU rd, rs, rt

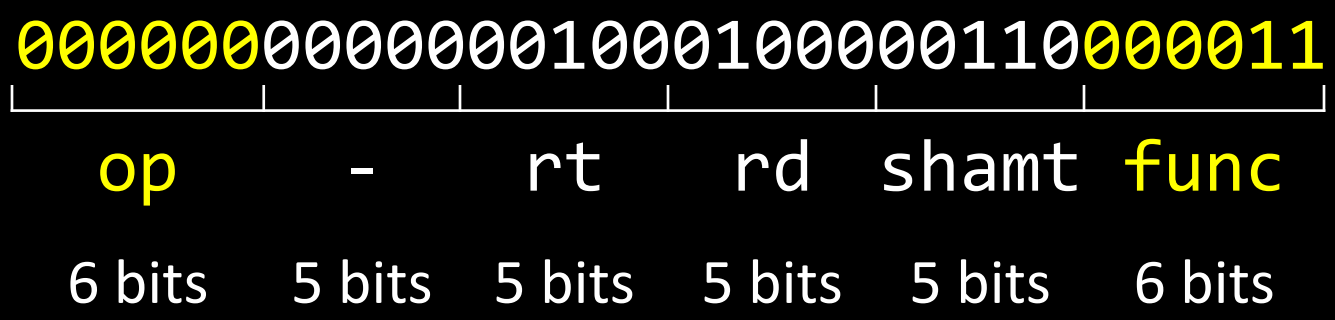
OR rd, rs, rt

XOR rd, rs, rt

NOR rd, rs rt


Instruction fetch + decode + ALU
= Babbage's engine + speed + reliability – hand crank

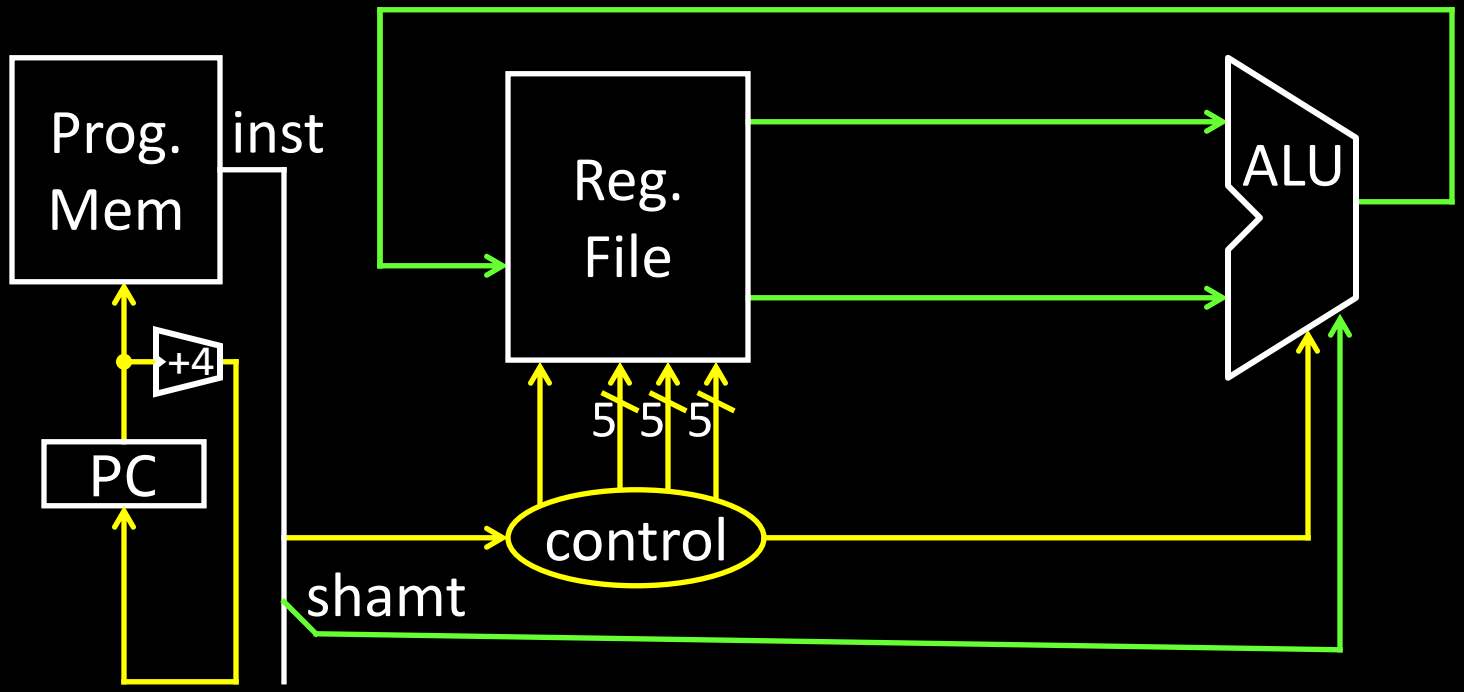
# Arithmetic Instructions: Shift



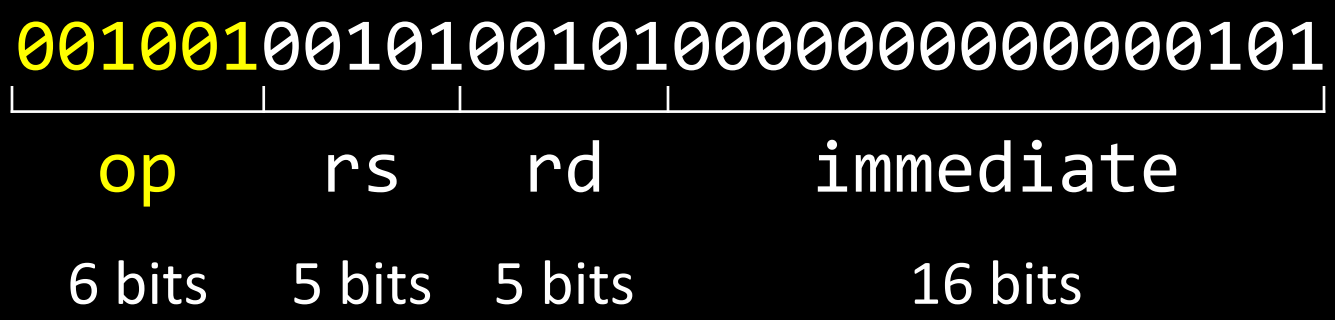

| op | func | mnemonic | description |
|---|---|---|---|
| 0x0 | 0x0 | SLL rd, rs, shamt | R[rd] = R[rt] << shamt |
| 0x0 | 0x2 | SRL rd, rs, shamt | R[rd] = R[rt] >>> shamt (zero ext.) |
| 0x0 | 0x3 | SRA rd, rs, shamt | R[rd] = R[rs] >> shamt (sign ext.) |

ex: r5 = r3 * 8

# Shift



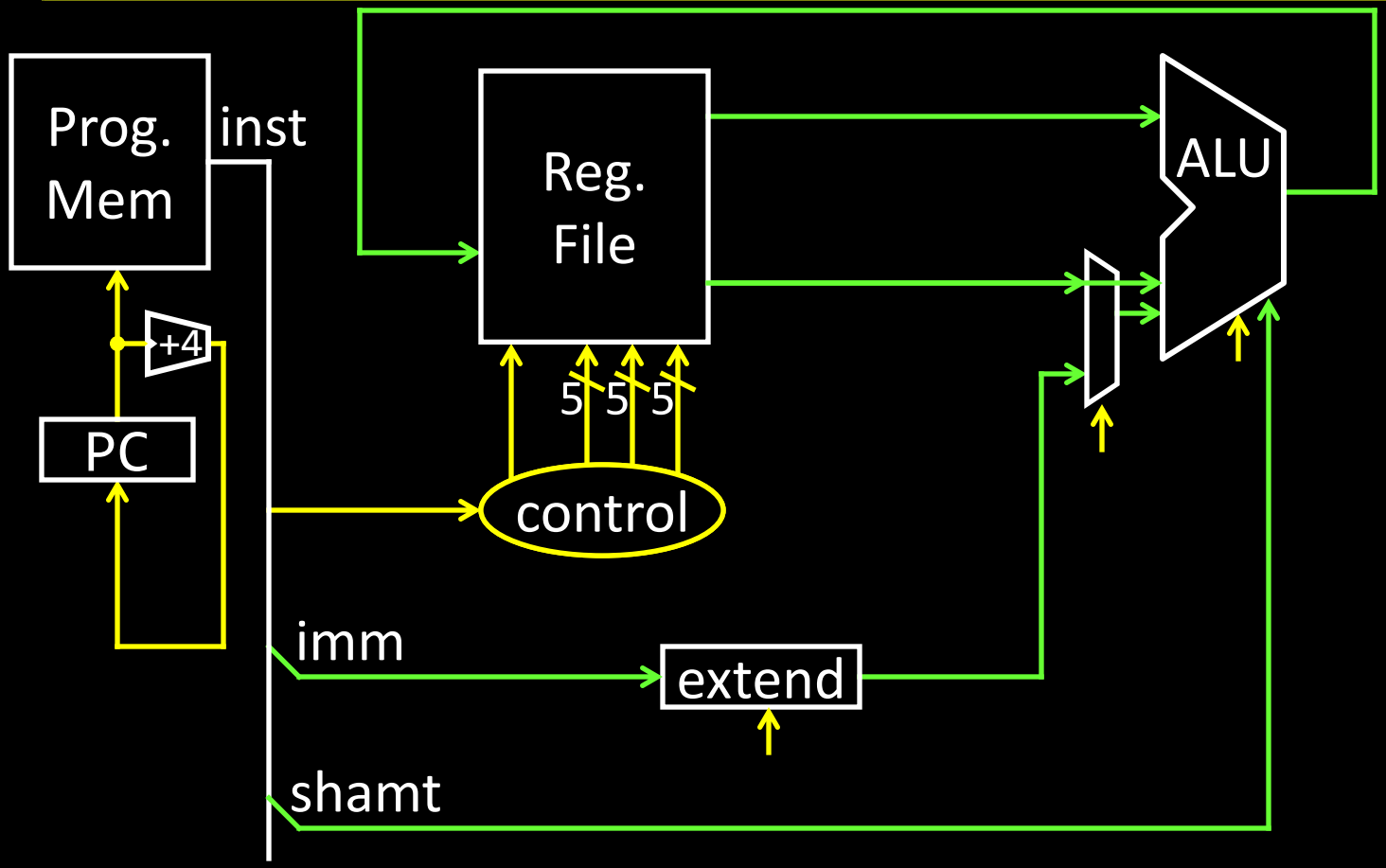
# Arithmetic Instructions: Immediates



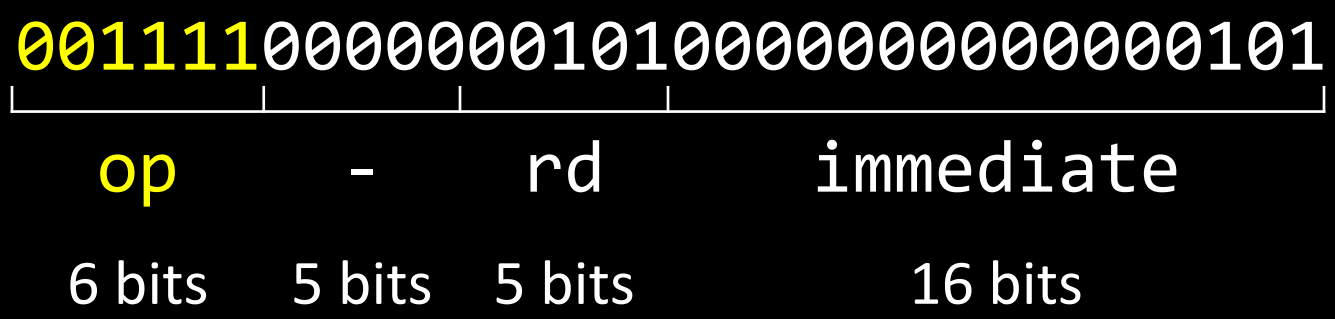

| op | mnemonic | description |
|---|---|---|
| 0x9 | ADDIU rd, rs, imm | R[rd] = R[rs] + sign_extend(imm) |
| 0xc | ANDI rd, rs, imm | R[rd] = R[rs] & zero_extend(imm) |
| 0xd | ORI rd, rs, imm | R[rd] = R[rs] \| zero_extend(imm) |

ex: r5 += 5 ex: r9 = -1 ex: r9 = 65535

# Immediates



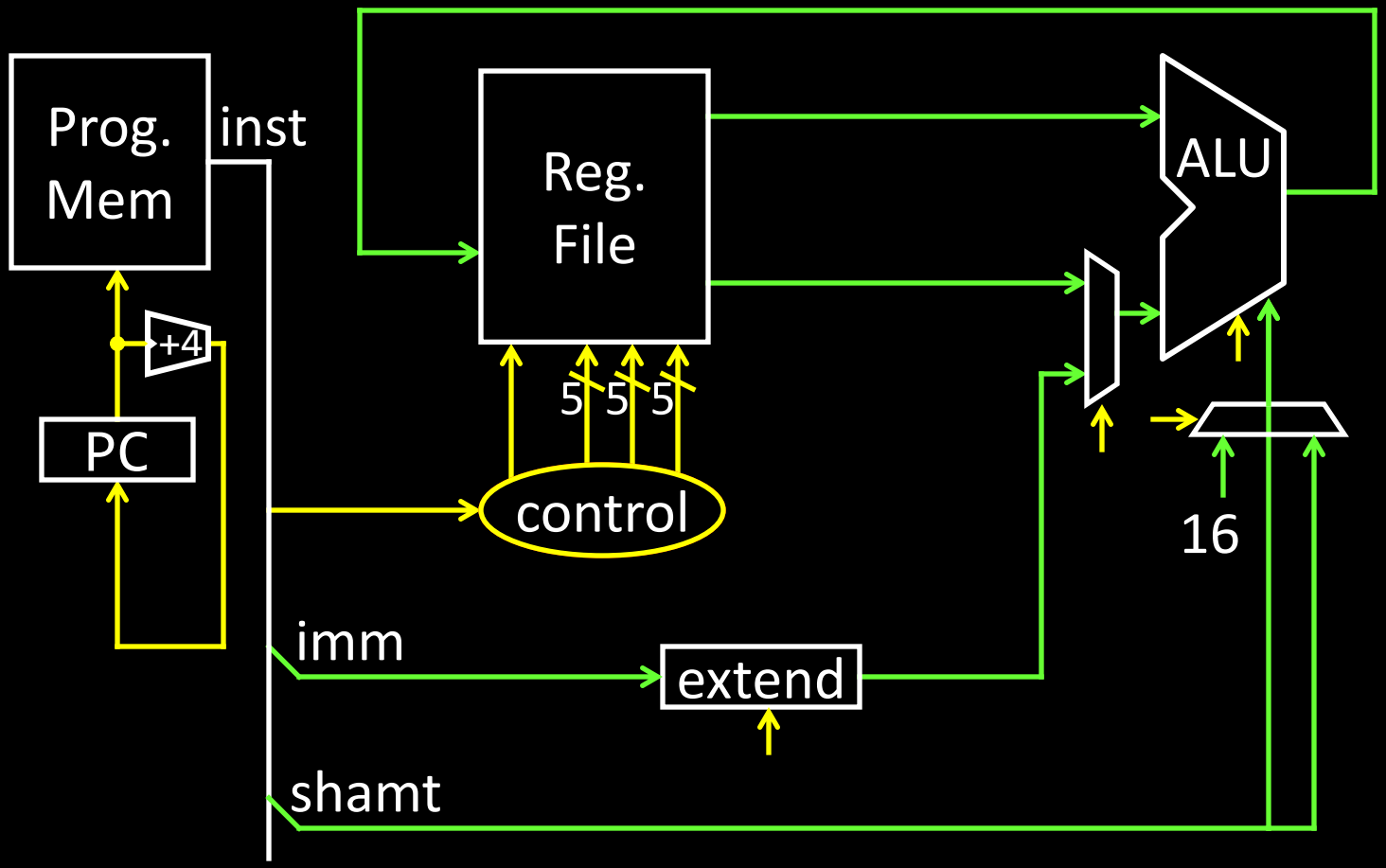
# Arithmetic Instructions: Immediates




| op | mnemonic | description |
|---|---|---|
| 0xF | LUI rd, imm | R[rd] = imm << 16 |

ex: r5 = 0xdeadbeef

# Immediates


Prog.
Mem
inst
Reg.
File
ALU
+4
PC
5
5
5
control
16
imm
extend
shamt

# MIPS Instruction Types

## Arithmetic/Logical

- R-type: result and two source registers, shift amount
- I-type:  16-bit immediate with sign/zero extension

## Memory Access

- load/store between registers and memory
- word, half-word and byte operations

## Control flow

- conditional branches: pc-relative addresses
- jumps: fixed offsets, register absolute

# Memory Instructions

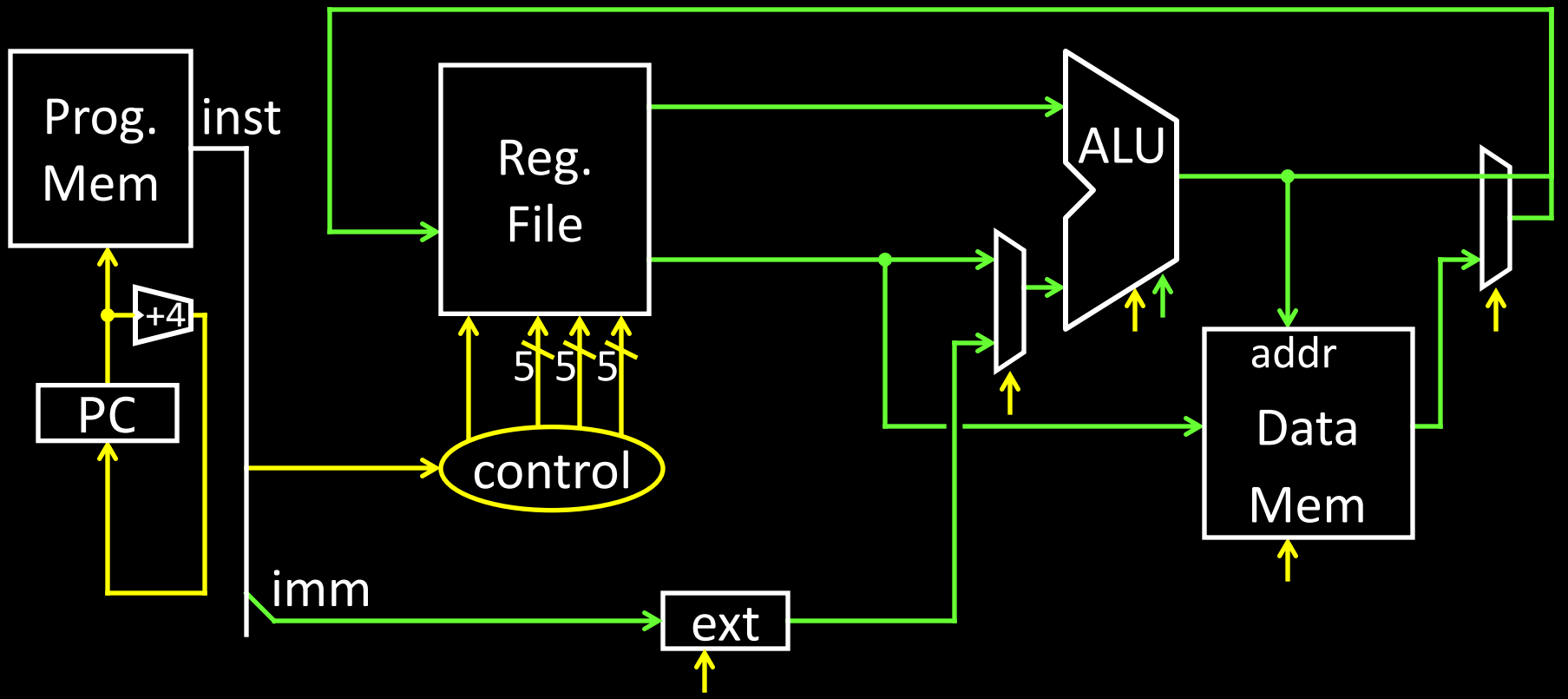

I-Type

base + offset addressing

| op | mnemonic | description |
|---|---|---|
| 0x20 | LB rd, offset(rs) | R[rd] = sign_ext(Mem[offset+R[rs]]) |
| 0x24 | LBU rd, offset(rs) | R[rd] = zero_ext(Mem[offset+R[rs]]) |
| 0x21 | LH rd, offset(rs) | R[rd] = sign_ext(Mem[offset+R[rs]]) |
| 0x25 | LHU rd, offset(rs) | R[rd] = zero_ext(Mem[offset+R[rs]]) |
| 0x23 | LW rd, offset(rs) | R[rd] = Mem[offset+R[rs]] |
| 0x28 | SB rd, offset(rs) | Mem[offset+R[rs]] = R[rd] |
| 0x29 | SH rd, offset(rs) | Mem[offset+R[rs]] = R[rd] |
| 0x2b | SW rd, offset(rs) | Mem[offset+R[rs]] = R[rd] |

signed offsets

# Memory Operations



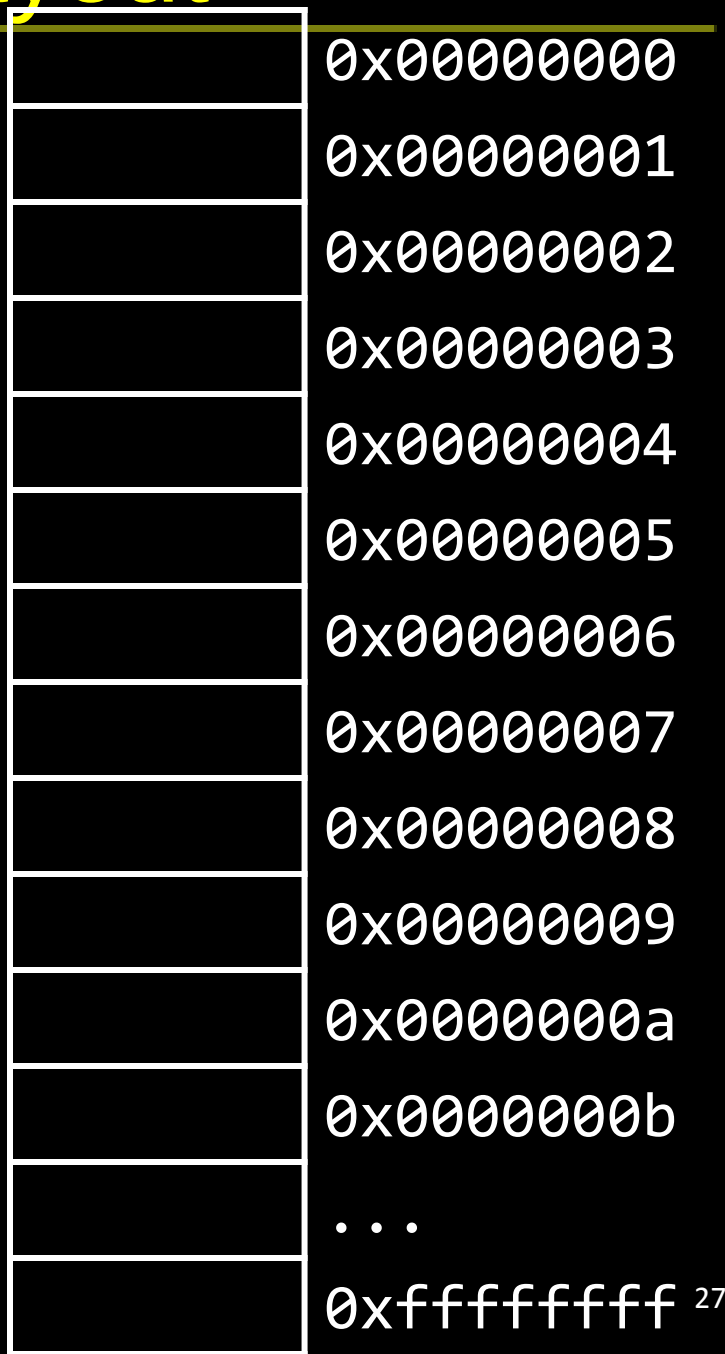
# Example

```
int h, A[];
A[12] = h + A[8];
```

# Memory Layout

Examples:

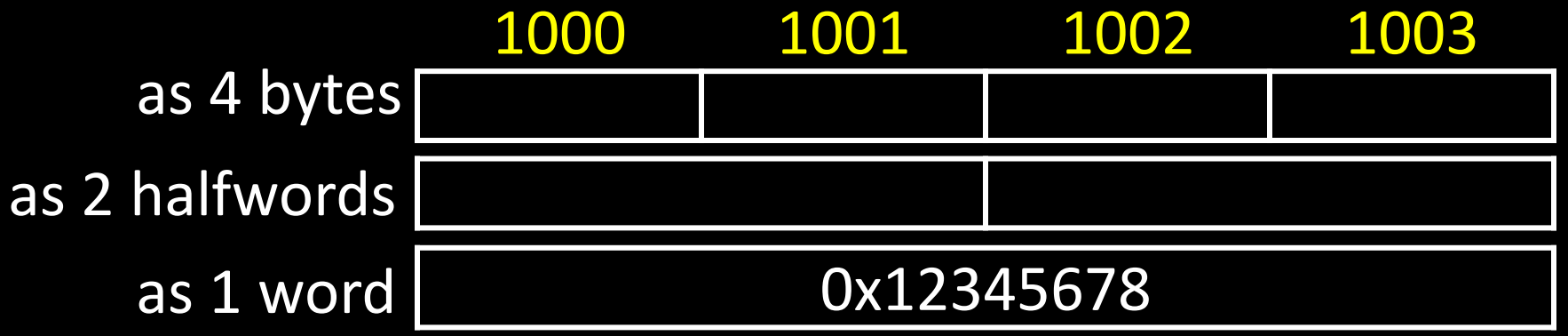
```
# r5 contains 0x5
sb r5, 2(r0)
lb r6, 2(r0)

sw r5, 8(r0)
lb r7, 8(r0)
lb r8, 11(r0)
```



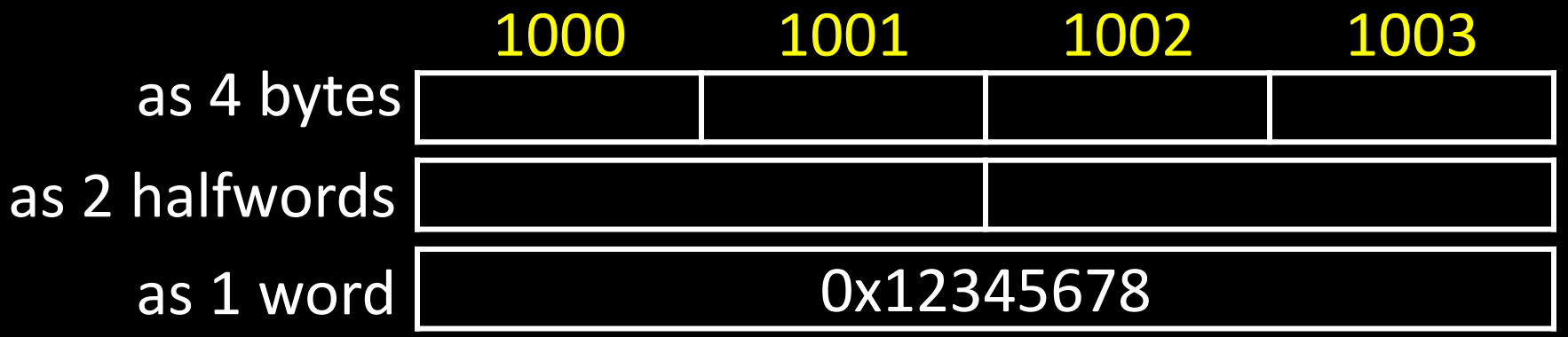
# Endianness

Endianness: Ordering of bytes within a memory word

Little Endian = least significant part first (MIPS, x86)




Big Endian = most significant part first (MIPS, networks)



# Control Flow: Absolute Jump


J-Type

| op | mnemonic | description |
|---|---|---|
| 0x2 | J target | PC = $(PC+4)_{32..29}$ \|\| target \|\| 00 |

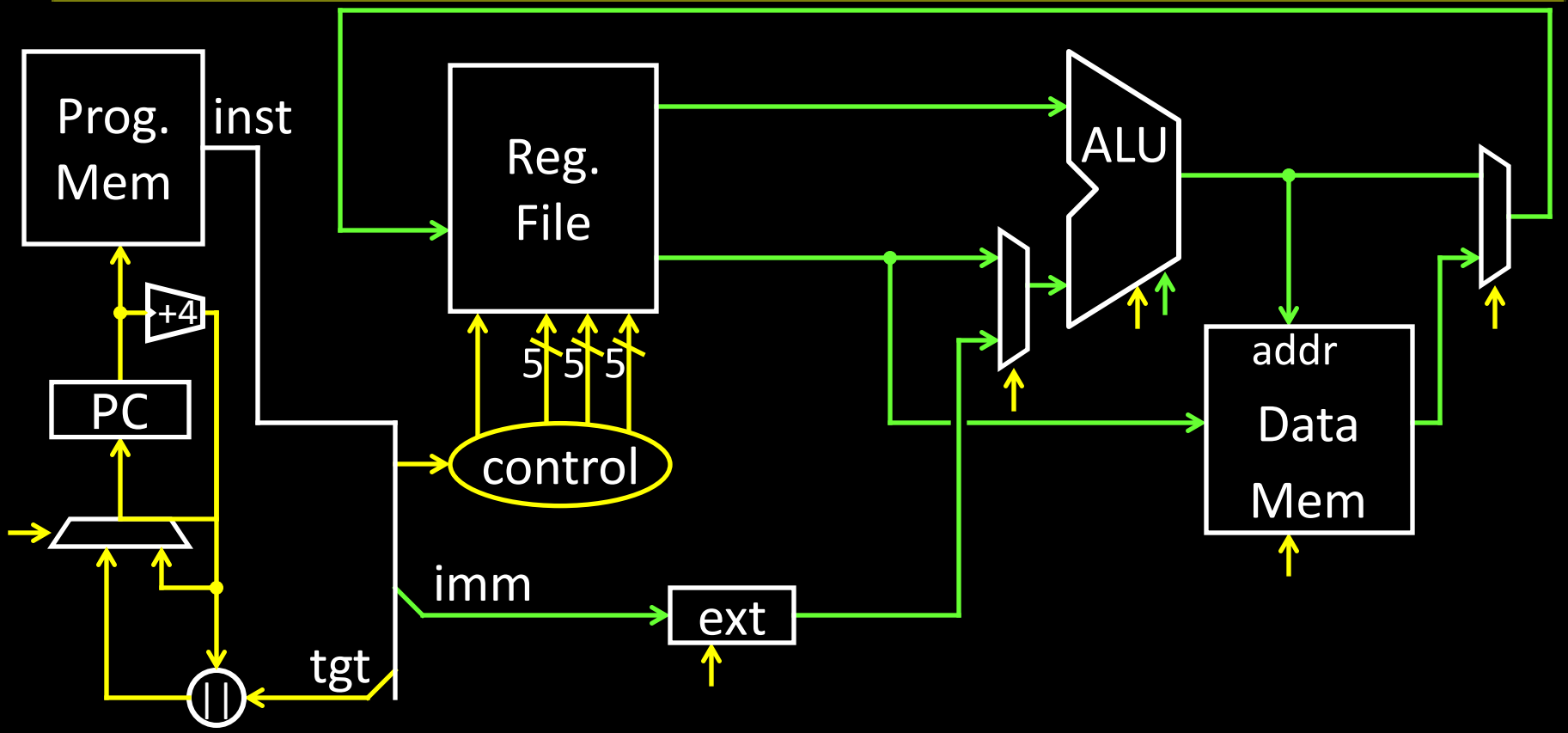
## Absolute addressing for jumps

- Jump from 0x30000000 to 0x20000000?
  - But: Jumps from 0x2FFFFFFF to 0x3xxxxxxx are possible, but not reverse
- Trade-off: out-of-region jumps vs. 32-bit instruction encoding

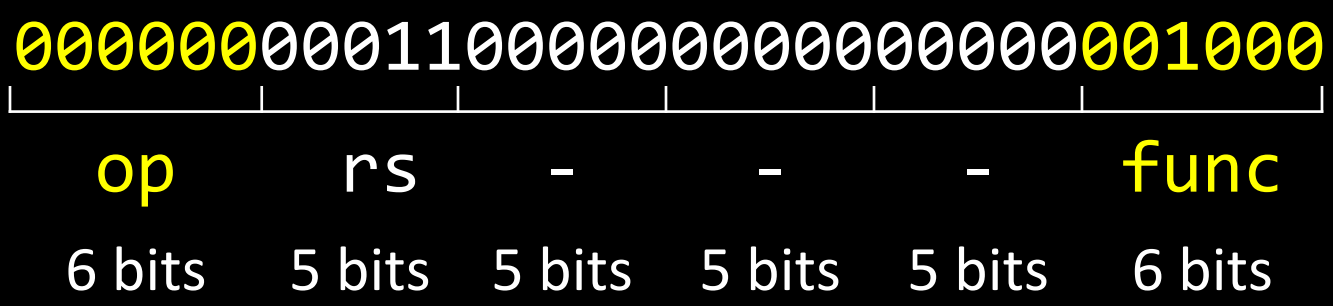
## MIPS Quirk:

- jump targets computed using *already incremented* PC

# Absolute Jump

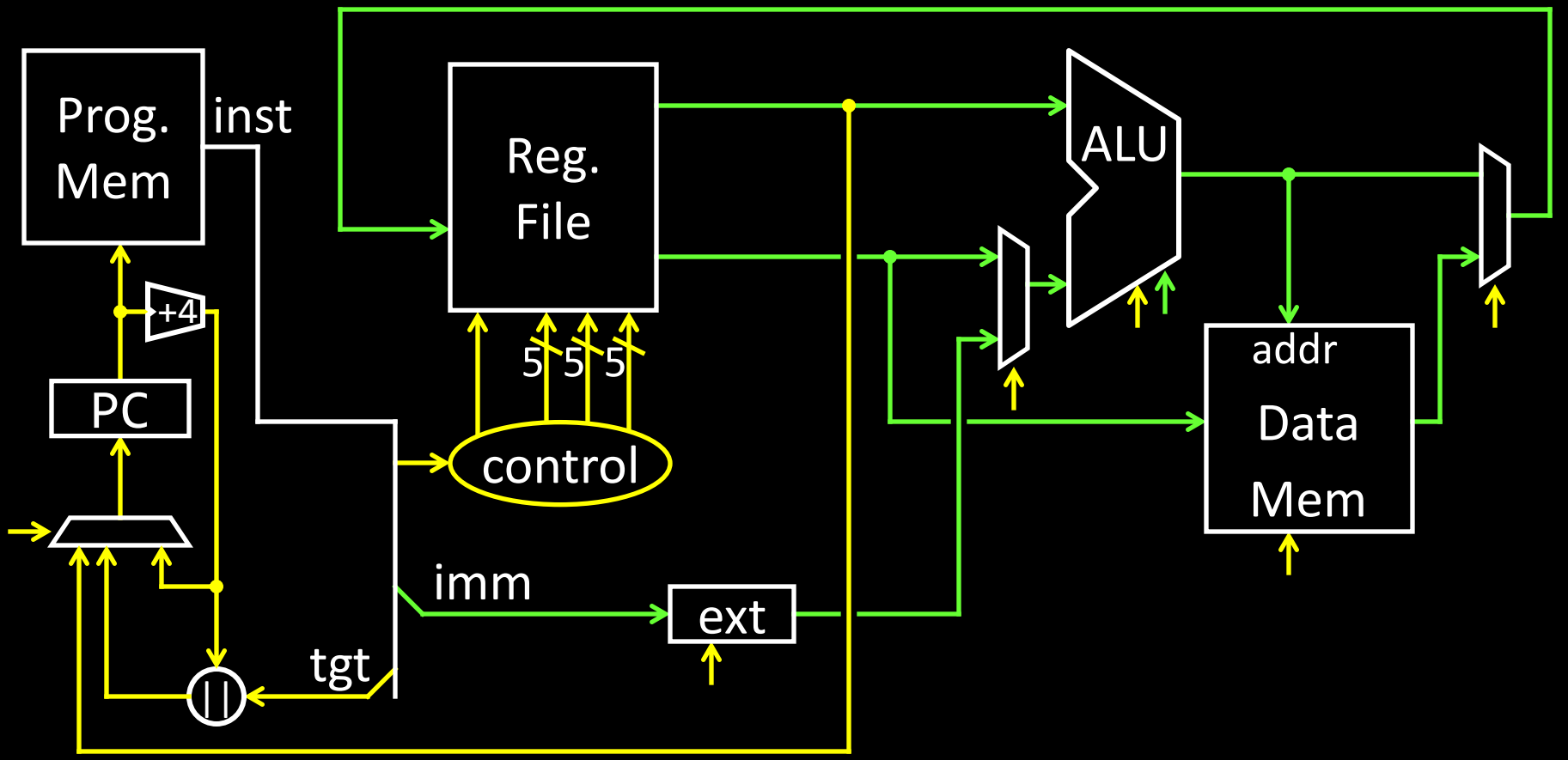


# Control Flow: Jump Register




| op | func | mnemonic | description |
|---|---|---|---|
| 0x0 | 0x08 | JR rs | PC = R[rs] |

# Jump Register



# Examples (2)

jump to 0xabcd1234

# Examples (2)

jump to 0xabcd1234

# assume 0 <= r3 <= 1

if (r3 == 0) jump to 0xdecafe0

else jump to 0xabcd1234

# Examples (2)

jump to 0xabcd1234

# assume 0 <= r3 <= 1

if (r3 == 0) jump to 0xdecafe0

else jump to 0xabcd1234

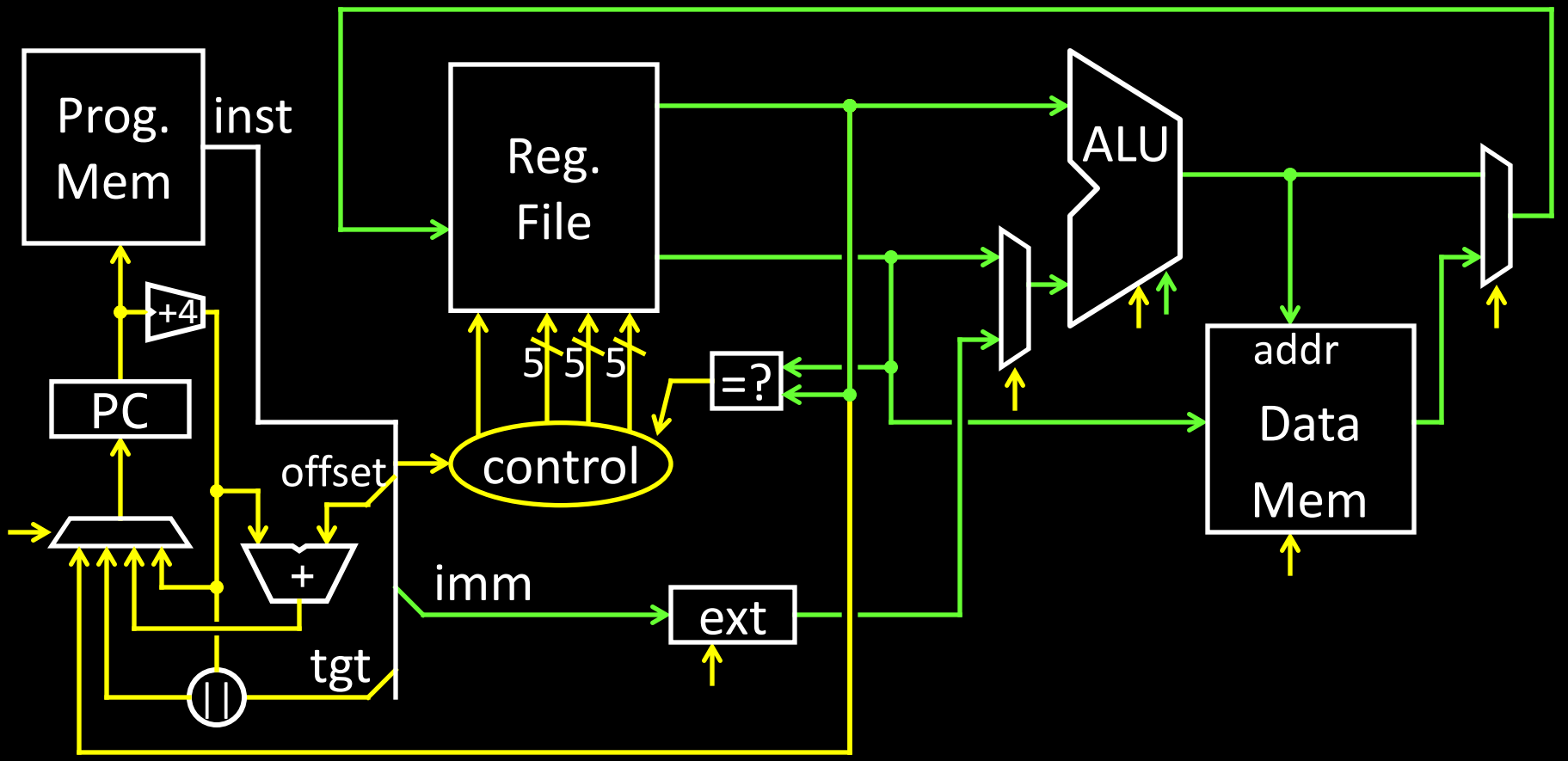
# Control Flow: Branches


| op | mnemonic | description |
|---|---|---|
| 0x4 | BEQ rs, rd, offset | if R[rs] == R[rd] then PC = PC+4 + (offset<<2) |
| 0x5 | BNE rs, rd, offset | if R[rs] != R[rd] then PC = PC+4 + (offset<<2) |

# Examples (3)

```
if (i == j) { i = i * 4; }
else { j = i - j; }
```

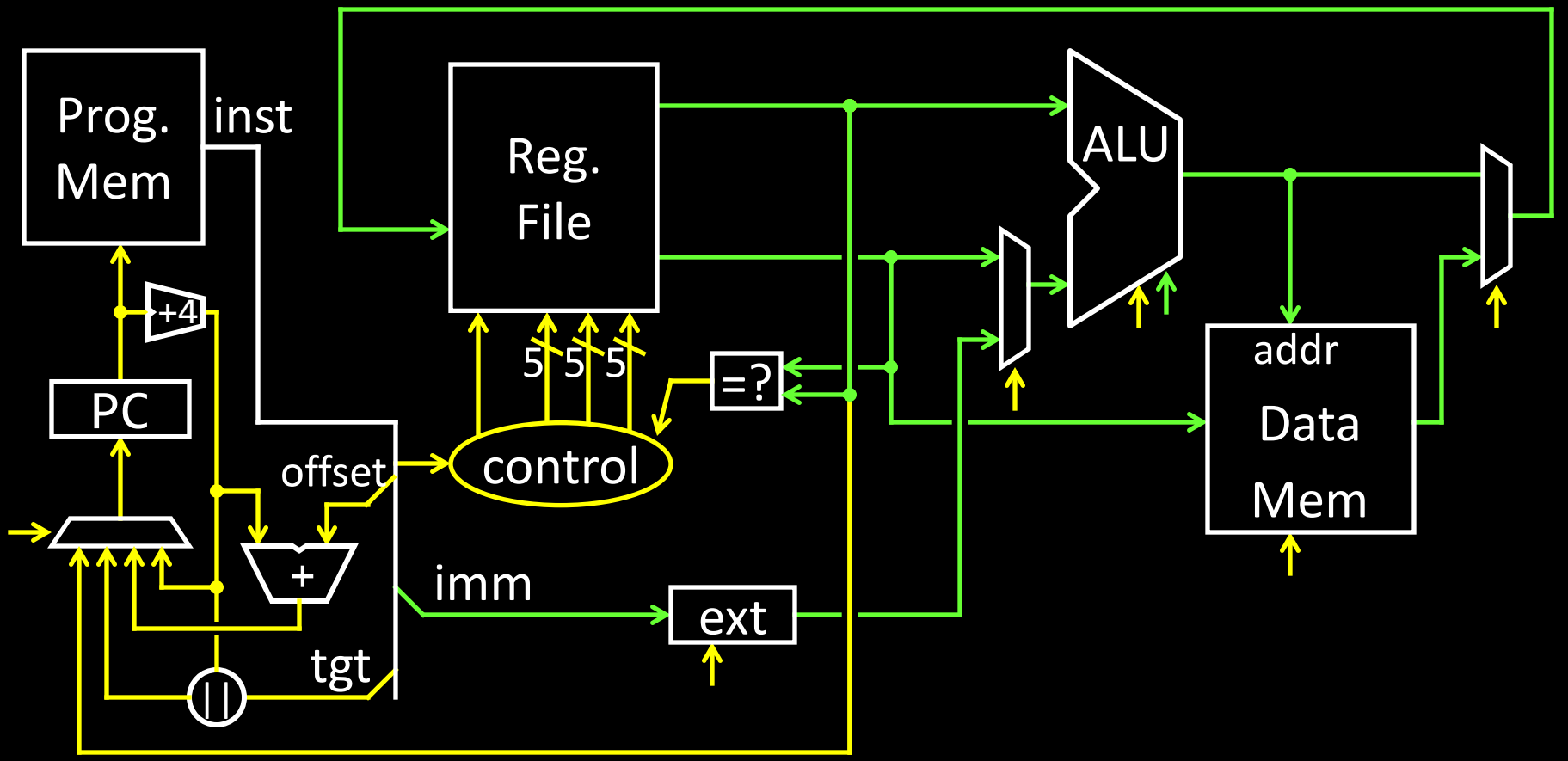
# Absolute Jump




**Could have used ALU for branch add**

**Could have used ALU for branch cmp**

# Absolute Jump




**Could have used ALU for branch add**

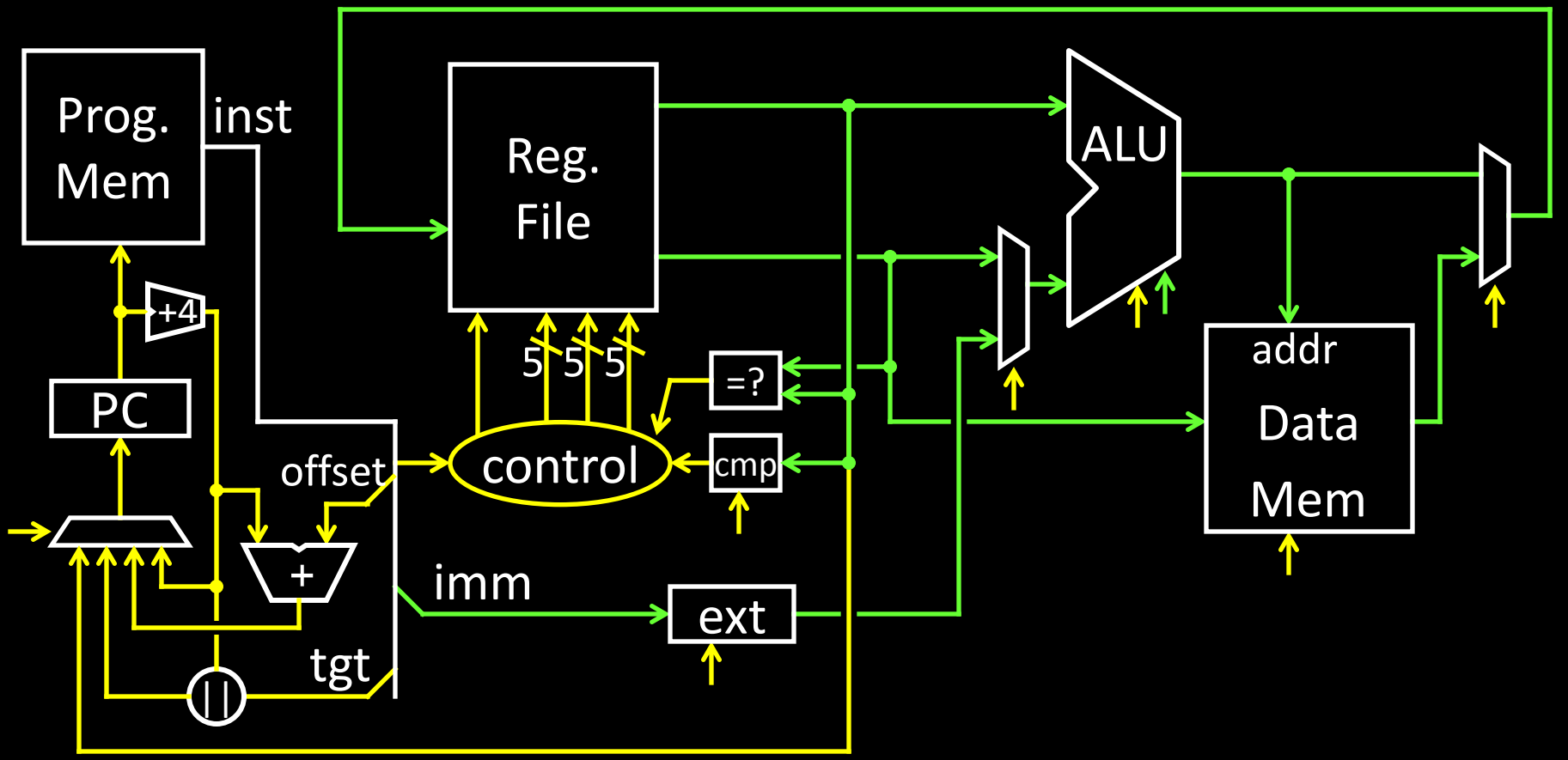
**Could have used ALU for branch cmp**

# Control Flow: More Branches


signed offsets

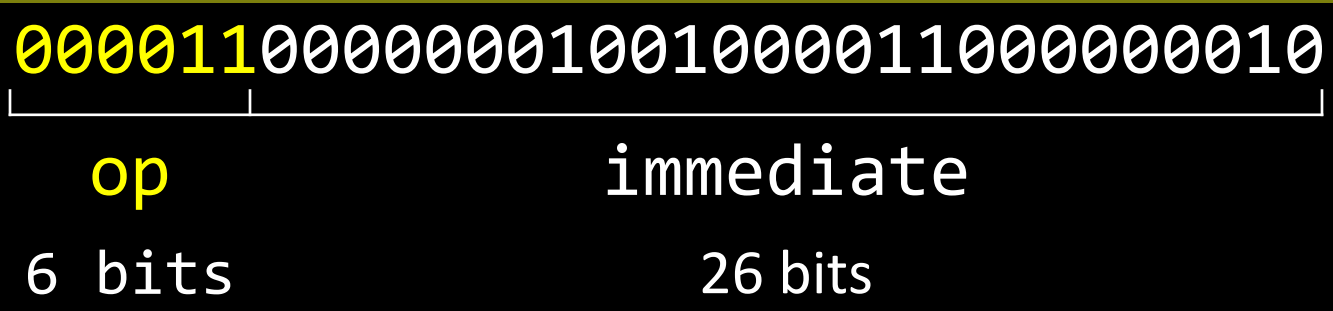
| op | subop | mnemonic | description |
|---|---|---|---|
| 0x1 | 0x0 | BLTZ rs, offset | if R[rs] < 0 then PC = PC+4+ (offset<<2) |
| 0x1 | 0x1 | BGEZ rs, offset | if R[rs] ≥ 0 then PC = PC+4+ (offset<<2) |
| 0x6 | 0x0 | BLEZ rs, offset | if R[rs] ≤ 0 then PC = PC+4+ (offset<<2) |
| 0x7 | 0x0 | BGTZ rs, offset | if R[rs] > 0 then PC = PC+4+ (offset<<2) |

# Absolute Jump



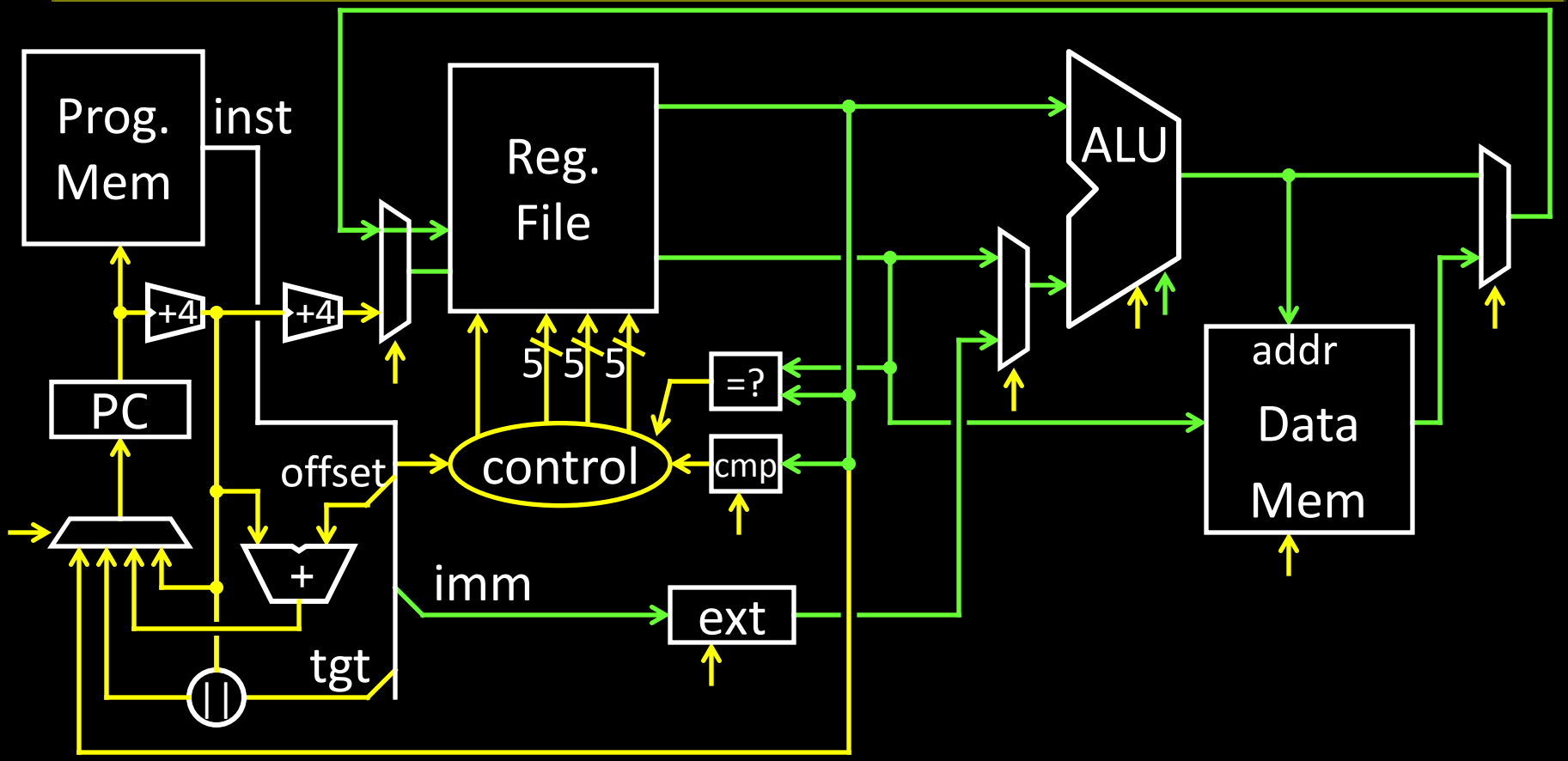

**Could have used ALU for branch cmp**

# Control Flow: Jump and Link




| op | mnemonic | description |
|---|---|---|
| 0x3 | JAL target | r31 = PC+8<br>PC = $(PC+4)_{32..29}$ \|\| target \|\| 00 |

# Absolute Jump




**Could have used ALU for link add**

# Next Time

CPU Performance

Pipelined CPU