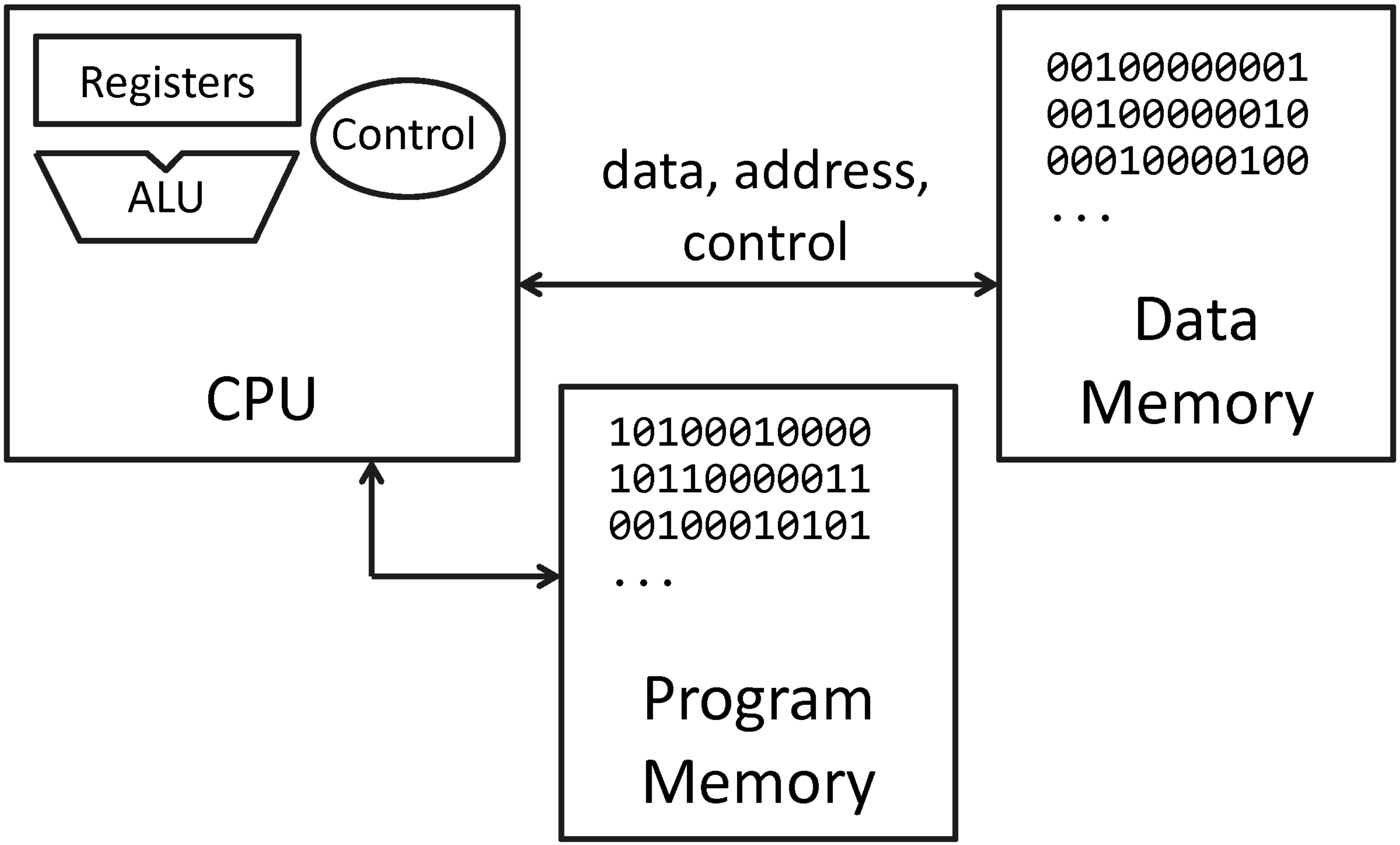
A Processor

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

See: P&H Chapter 2.16-20, 4.1-3

Let's build a MIPS CPU

- …but using Harvard architecture



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

High Level Language

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

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

Assembly Language

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

```
00100000000000110000000000000011
00100000000001010000000000000000
00010000100001010000000000000011
```

Machine Langauge

- Binary-encoded assembly
- Labels become addresses

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

MIPS = Reduced Instruction Set Computer (RISC)

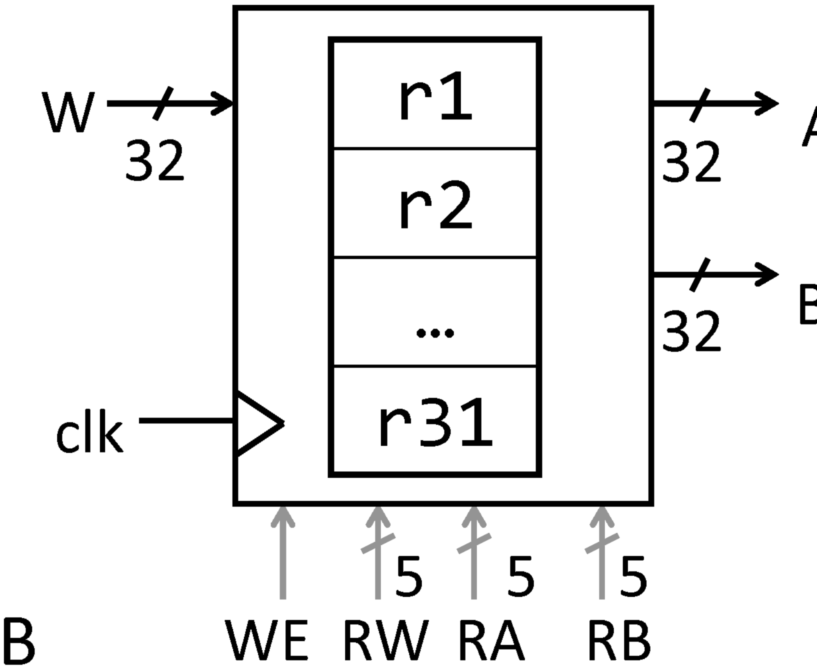
- ≈ 200 instructions, 32 bits each, 3 formats
  - mostly orthogonal
- all operands in registers
  - almost all are 32 bits each, can be used interchangeably
- ≈ 1 addressing mode: Mem[reg + imm]

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

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



MIPS Memory

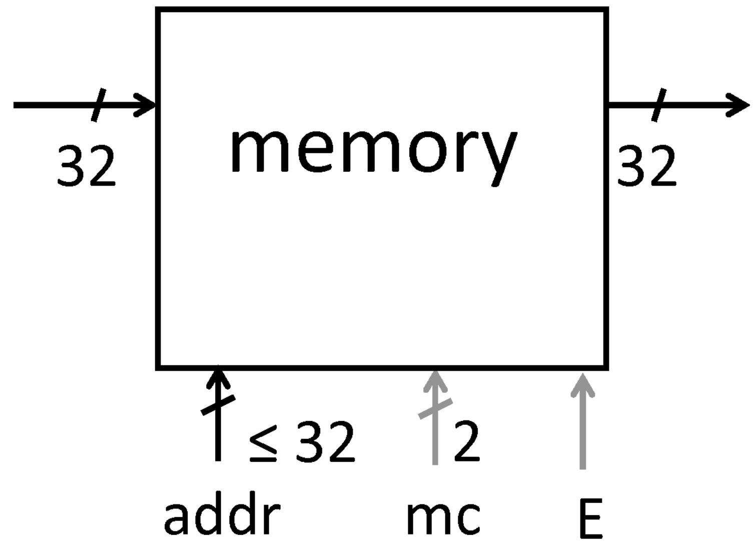
- Up to 32-bit address
- 32-bit data
  (but byte addressed)
- Enable + 2 bit memory control

  00: read word (4 byte aligned)

  01: write byte

  10: write halfword (2 byte aligned)

  11: write word (4 byte aligned)

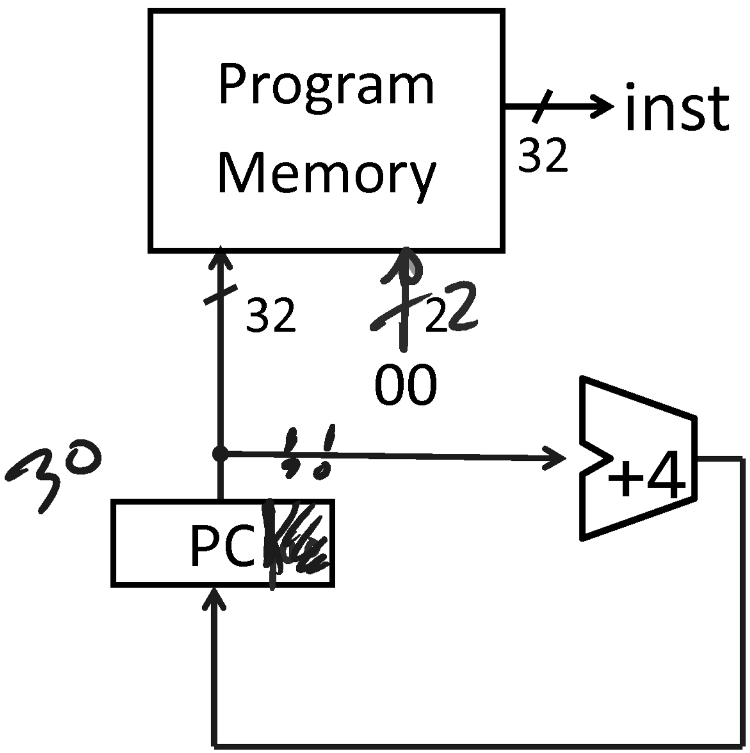


Basic CPU execution loop

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

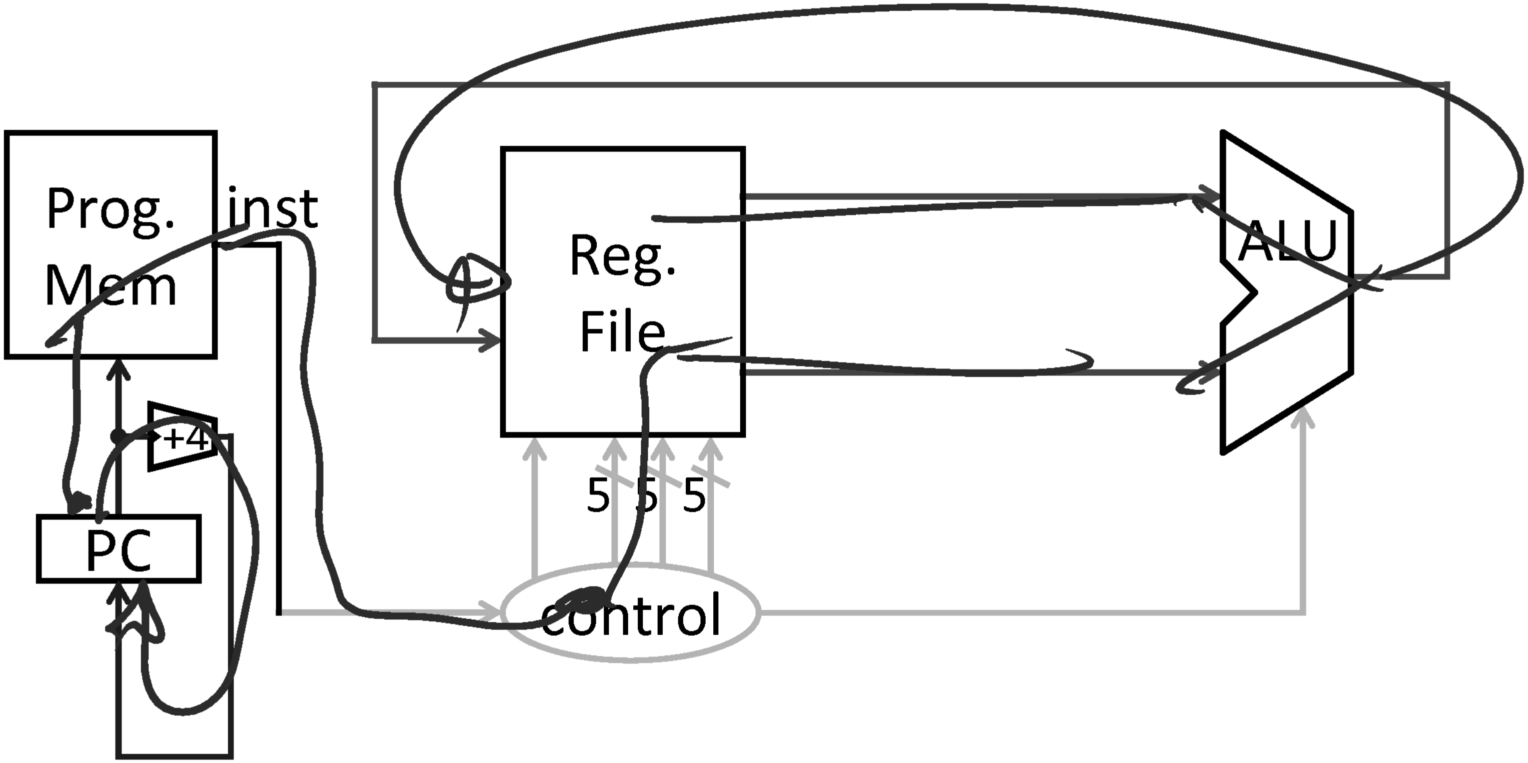
Instruction Fetch Circuit

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



000000	01000	00110	00100	00000	100110	
op	rs	rt	rd	-	func	**R-Type**
				0x2	6	
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

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])


Prog.
Mem
inst
Reg.
File
ALU
PC
+4
5
5
5
control

r4 = (r1 + r2) | r3

r8 = 4*r3 + r4 – 1

r9 = 9

ADDU rd, rs, rt

SUBU rd, rs, rt

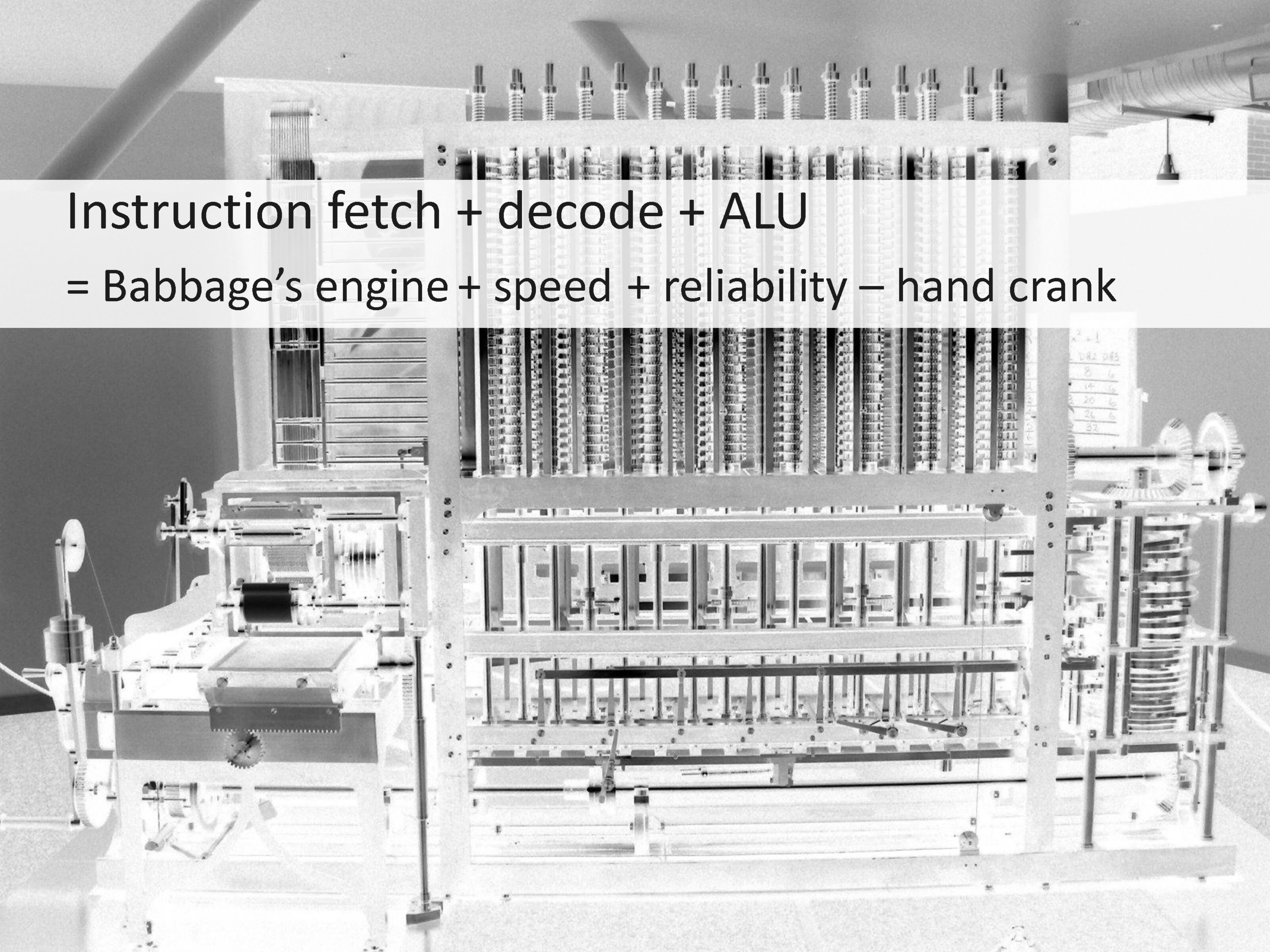
OR rd, rs, rt

XOR rd, rs, rt

NOR rd, rs rt

Instruction fetch + decode + ALU

= Babbage's engine + speed + reliability – hand crank

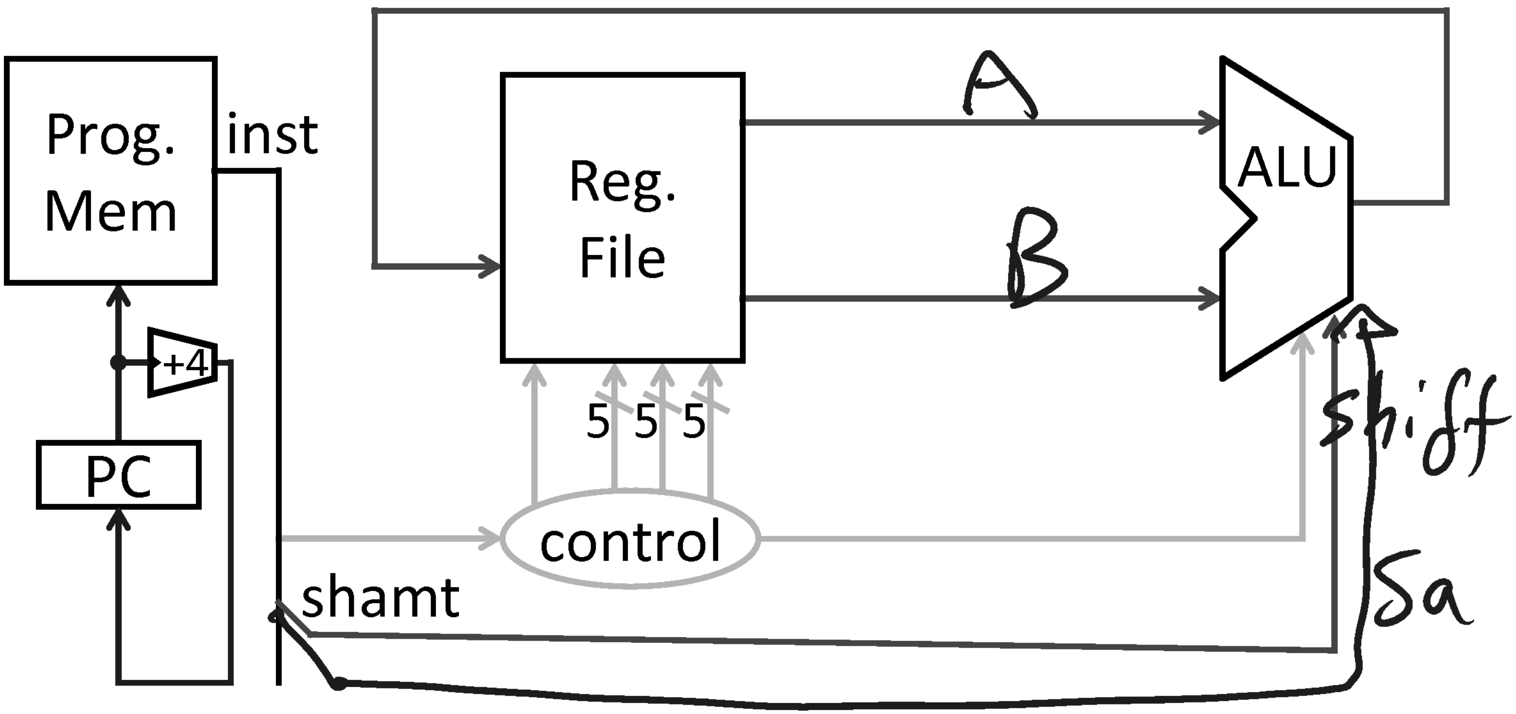


000000	00000	00010	00100	00110	000011
op	-	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-Type

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



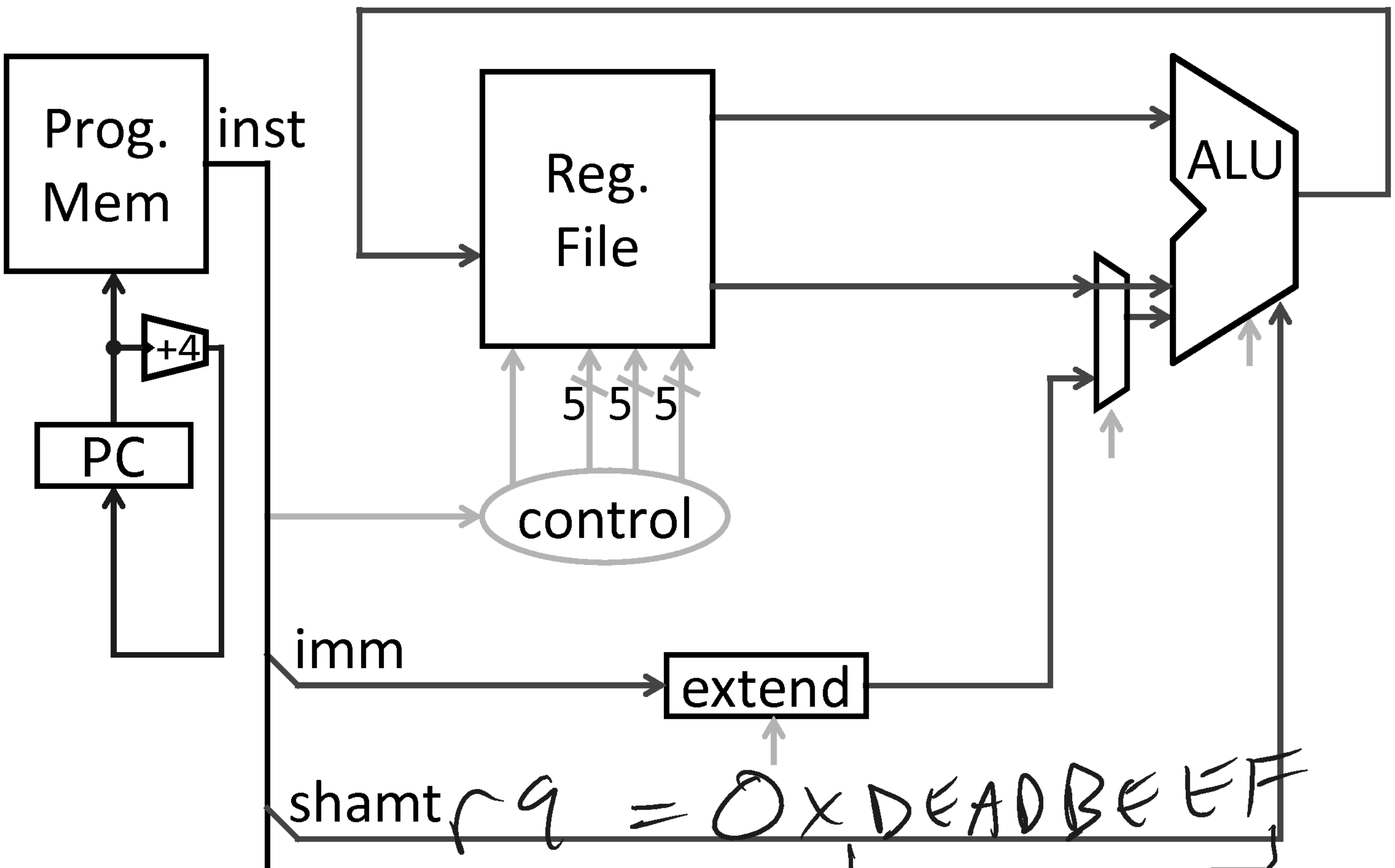
r3 = 4 * r5

001001	00101	00101	0000000000000101
op	rs	rd	immediate
	A		
6 bits	5 bits	5 bits	16 bits

I-Type

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



r9 = 0xDEADBEEF

ORI r9, r0, 0xDEAD
SLL r9, r9, 16
ORI r9, r0, 0xBEEF

`00111100000001010000000000000101`

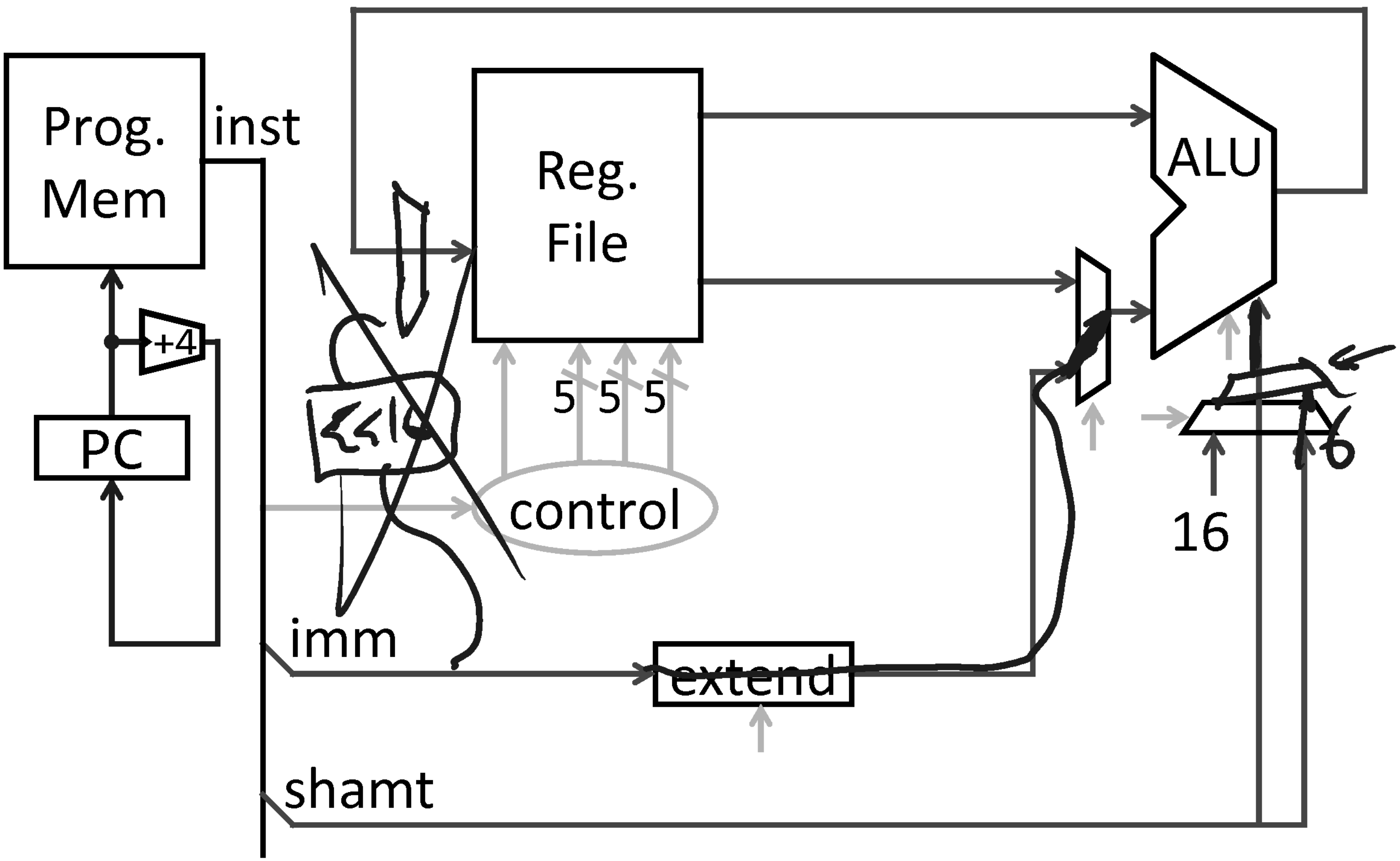
op	-	rd	immediate
001111	00000	00101	0000000000000101
6 bits	5 bits	5 bits	16 bits

I-Type

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

ex: r5 = 0xdeadbeef

Load upper imm.


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

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

101001 00101 00001 0000000000000010

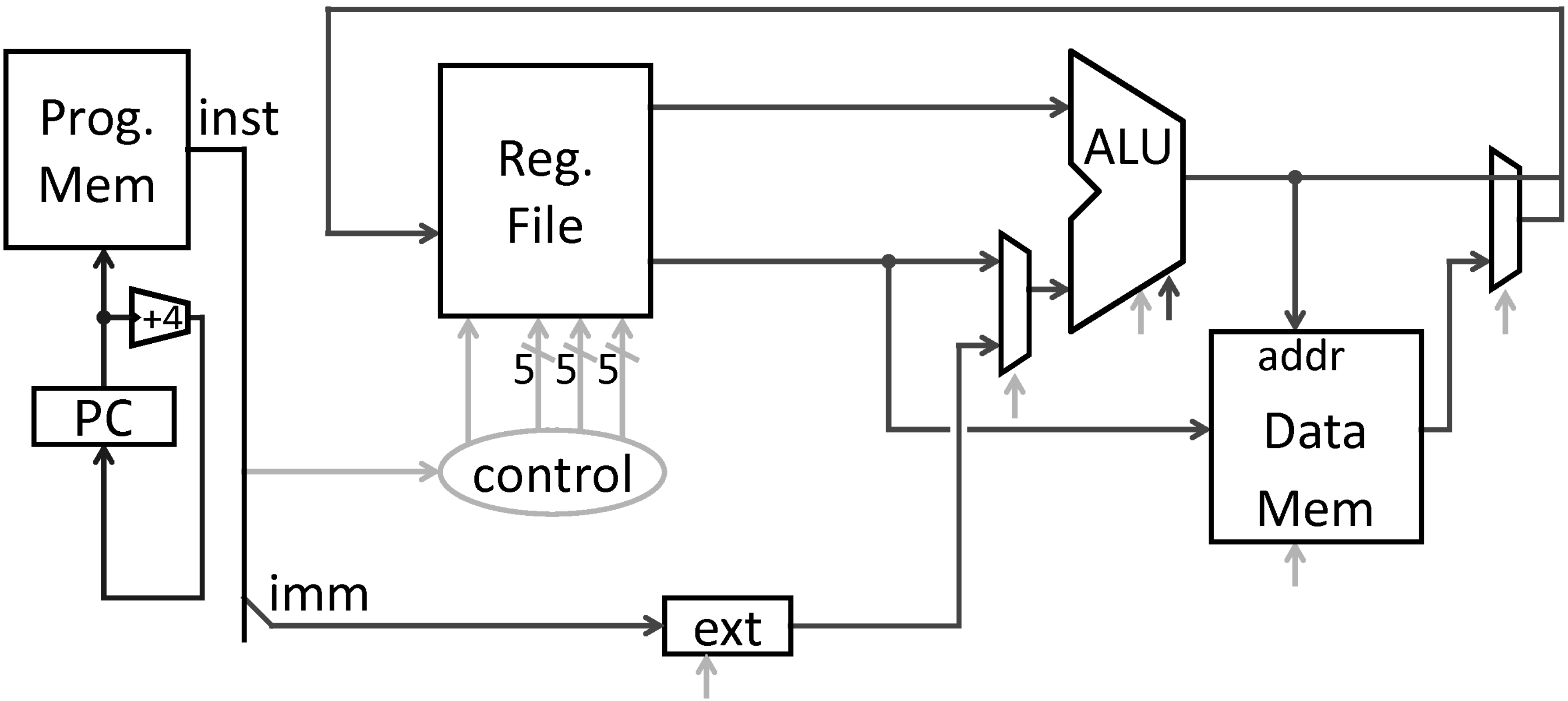
op	rs	rd	offset
6 bits	5 bits	5 bits	16 bits

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]

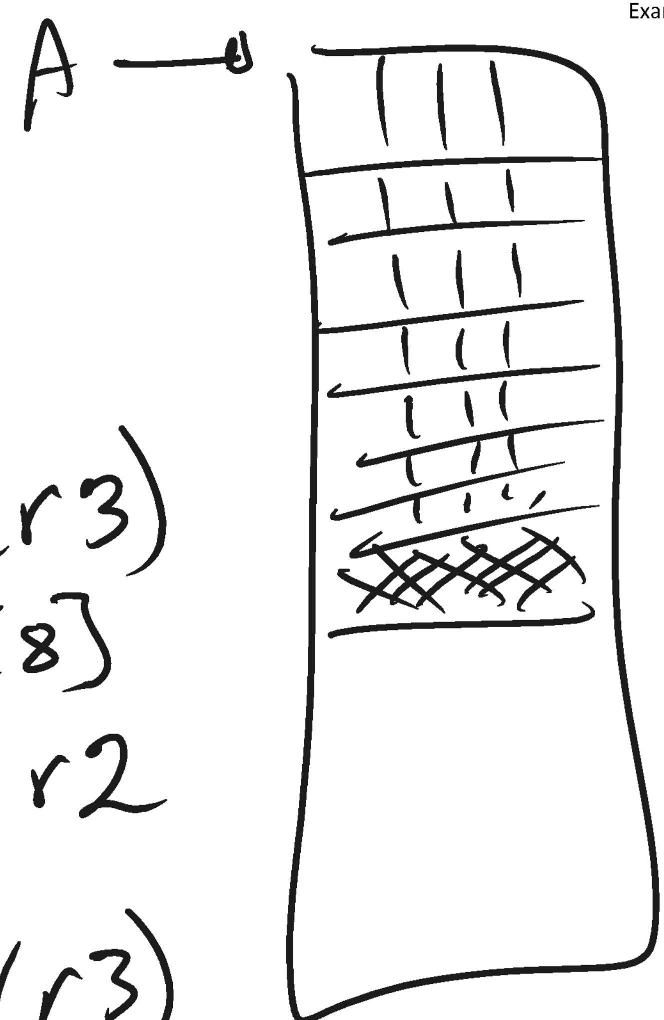
signed offsets



Prog. Mem
inst
Reg. File
ALU
+4
PC
5
5
5
control
addr
Data Mem
imm
ext

r2 r3

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




```
LW r4, 32(r3)
    # r4 = A[8]
ADD r5, r4, r2
SW r5, 48(r3)
```

Examples:

\# r5 contains ~~0xDEADBEEF~~ 5

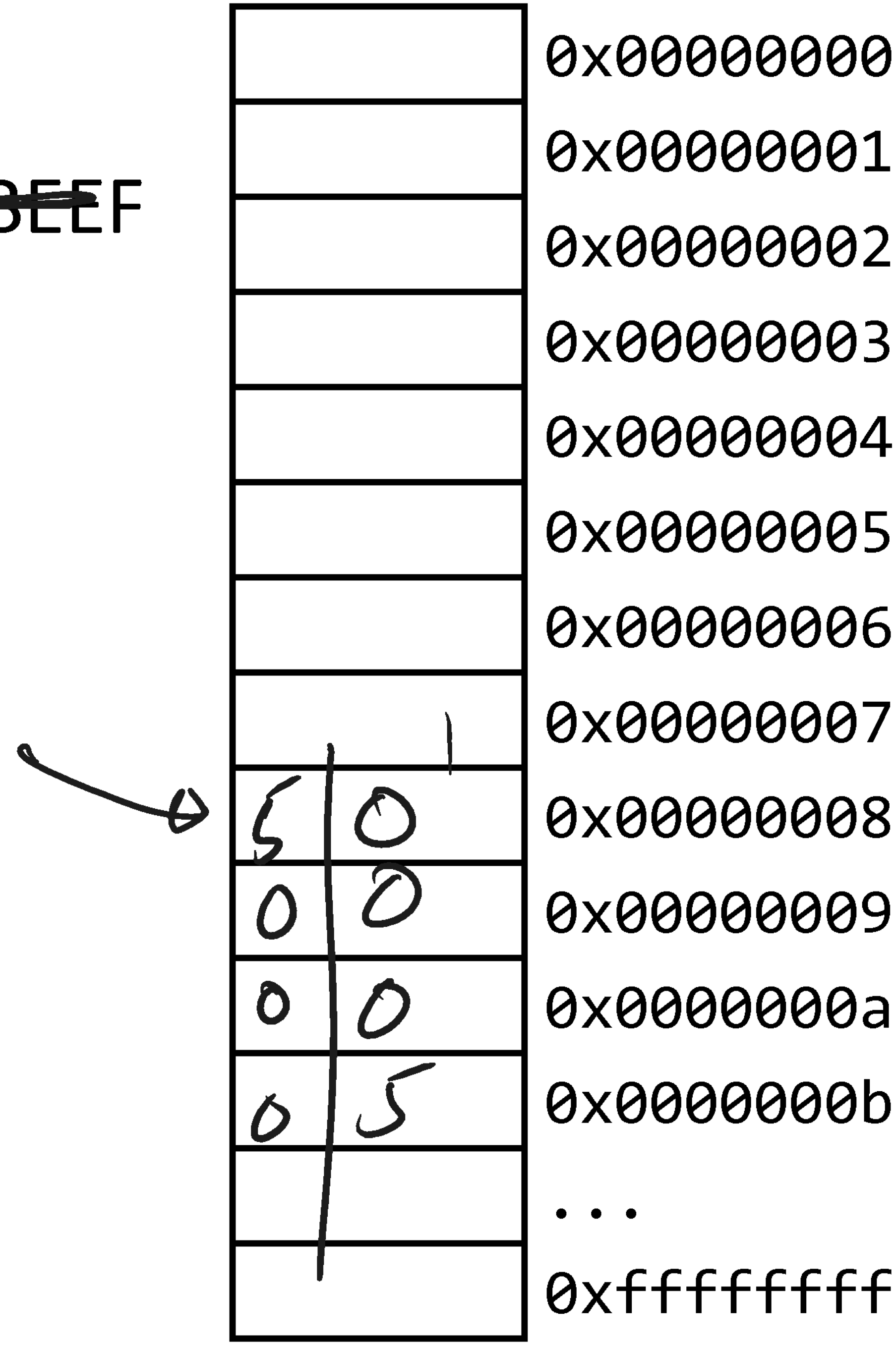
sb r5, 2(r0)

lb r6, 2(r0)

sw r5, 8(r0)

lb r7, 8(r0)

lb r8, 11(r0)



Endianness: Ordering of bytes within a memory word

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

	1000	1001	1002	1003
as 4 bytes	78	56	34	12
as 2 halfwords	5678		1234	
as 1 word	0x12345678			

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

	1000	1001	1002	1003
as 4 bytes	12	34	56	78
as 2 halfwords	1234		5678	
as 1 word	0x12345678			

```
000010 10100001001000011000000011
op     immediate
6 bits 26 bits
```

J-Type

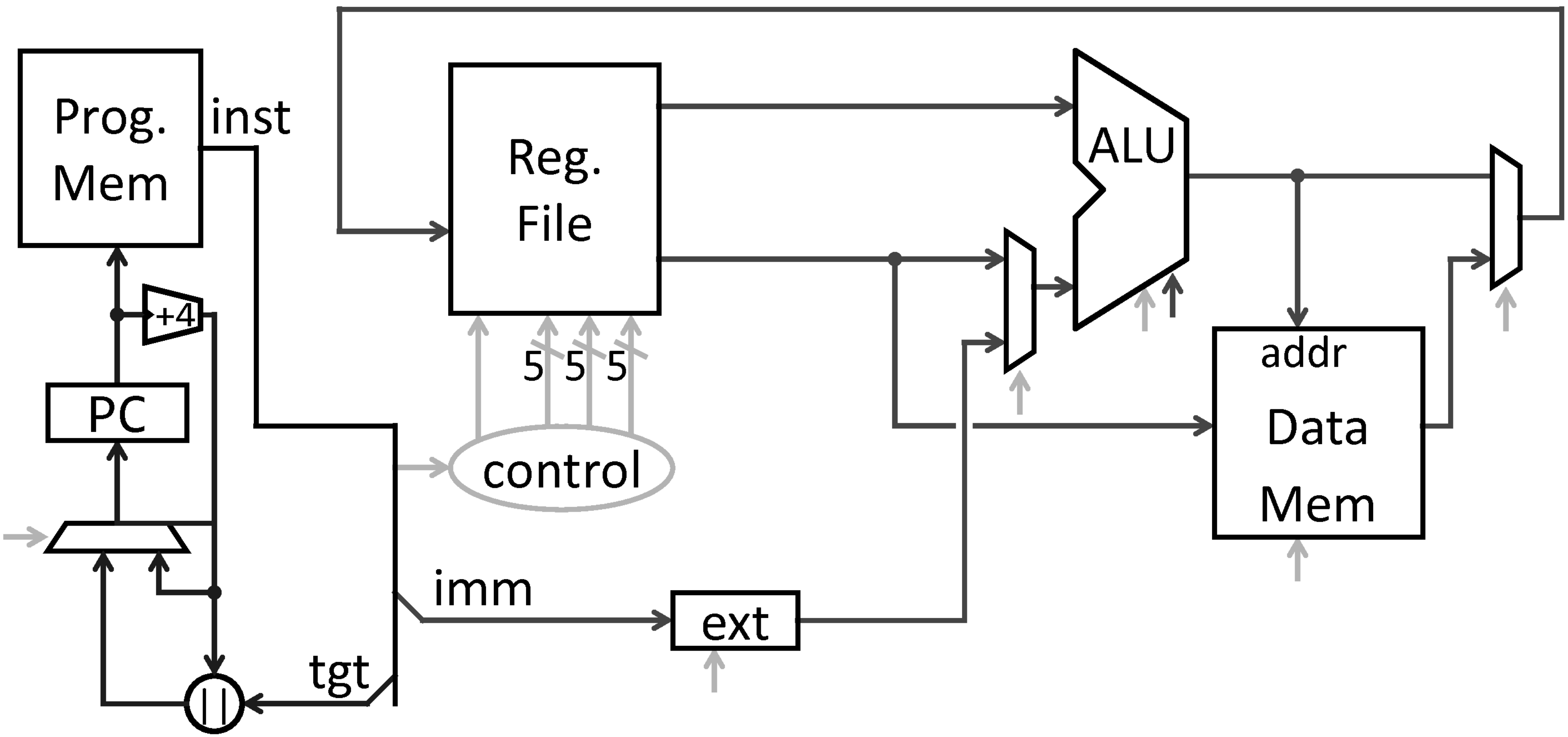
op	mnemonic	description
0x2	J target	PC = $(PC+4)_{32..29}$ || target || 00

Absolute addressing for jumps

- Jump from 0x30000000 to 0x20000000? NO Reverse? NO
  - 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

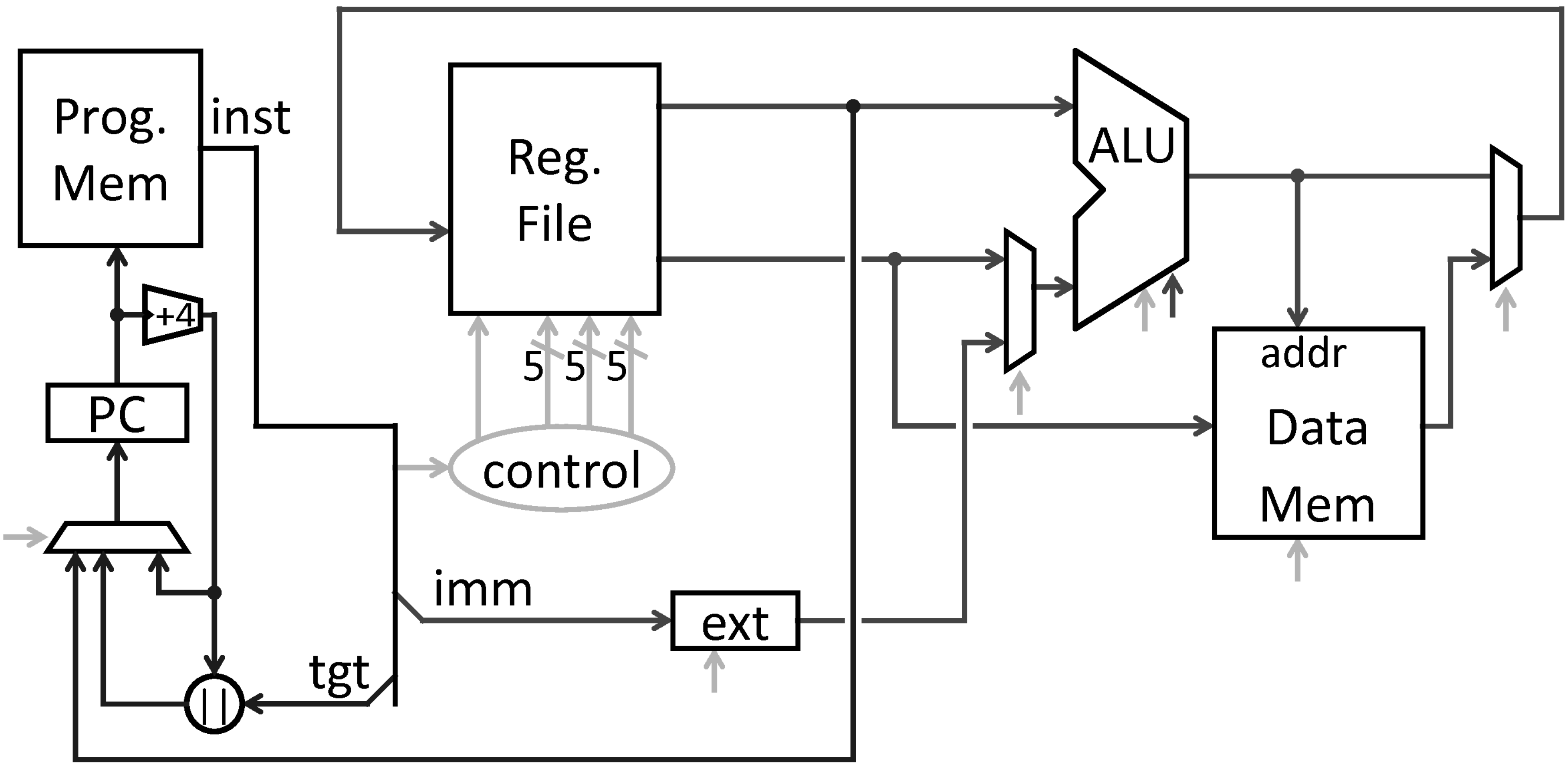


Prog. Mem
inst
Reg. File
ALU
5 5 5
control
addr
Data Mem
PC
+4
imm
ext
tgt
||

000000	00011	00000	00000	00000	001000
op	A rs	-	-	-	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-Type

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


Prog.
Mem
inst
Reg.
File
ALU
5 5 5
control
addr
Data
Mem
PC
+4
imm
ext
tgt
||

jump to 0xabcd123~~4~~5

~~J 0xabcd1234~~

```
LUI r3, 0xabcd
ORI r3, r3, 0xf23~~4~~5
JR  r3
```

\# assume 0 <= r3 <= 1

if (r3 == 0) jump to 0xdecafe0 = X

else jump to 0xabcd1234 = Y

r5 = Y - X

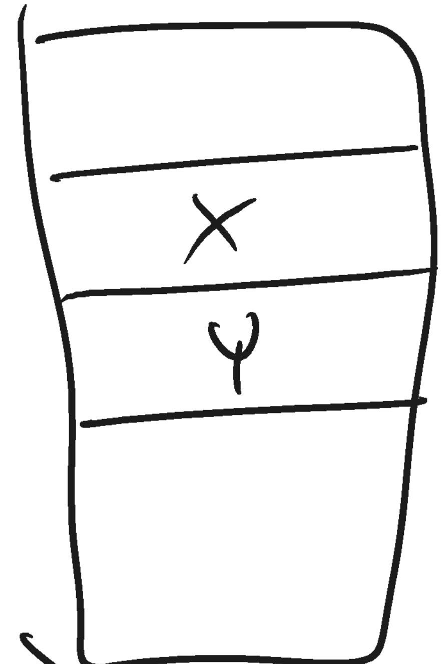
r5 = X + r3 * r5

JR r5

\# assume 0 <= r3 <= 1

if (r3 == 0) jump to 0xdecafe0 = x

else jump to 0xabcd1234 = y



r4 = x

sw r4, 0(r0)

r4 = y

sw r4, 4(r0)

000100	00101	00001	0000000000000011
op	rs	rd	offset
6 bits	5 bits	5 bits	16 bits

I-Type

signed offsets

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)

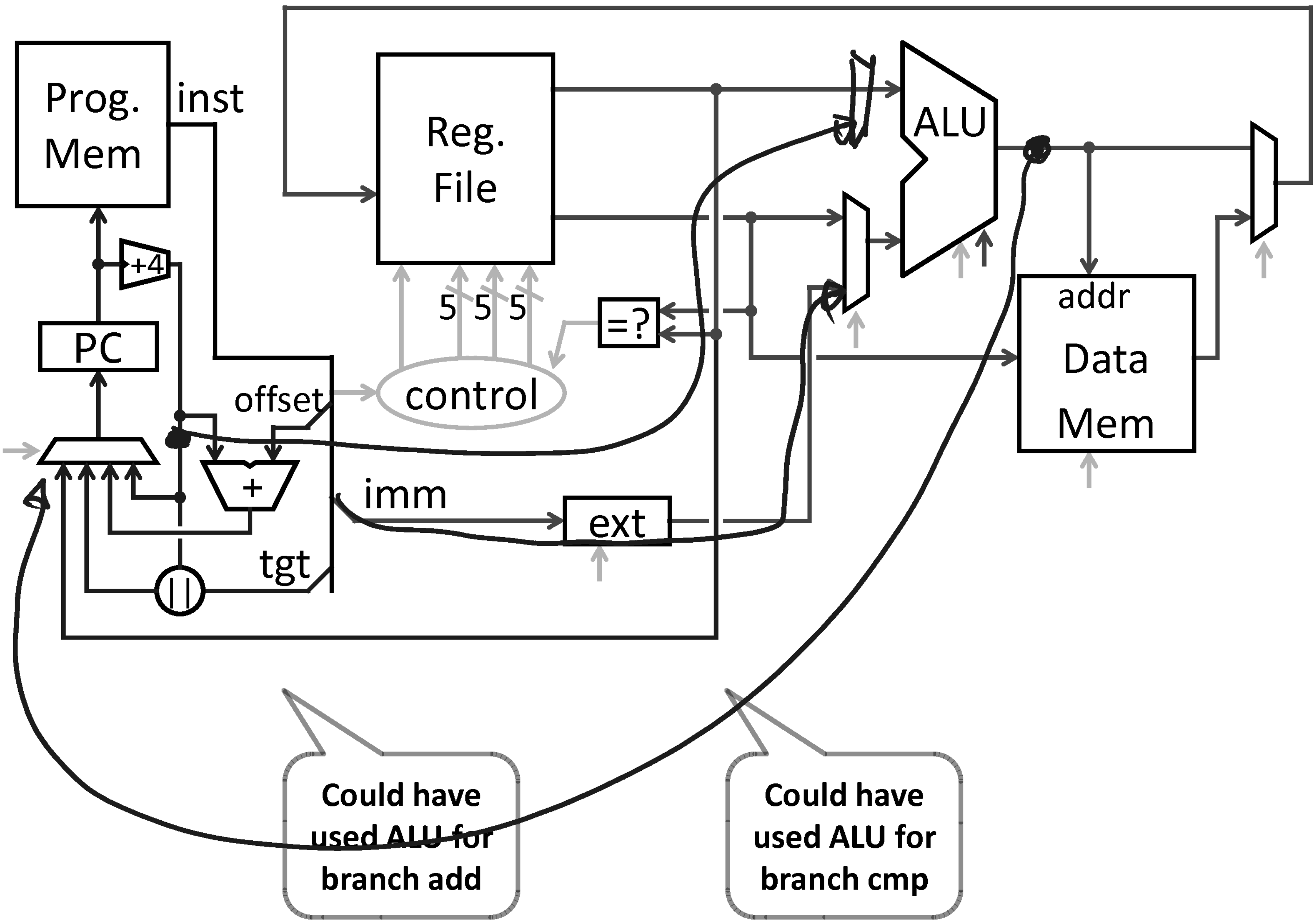
r5 r6

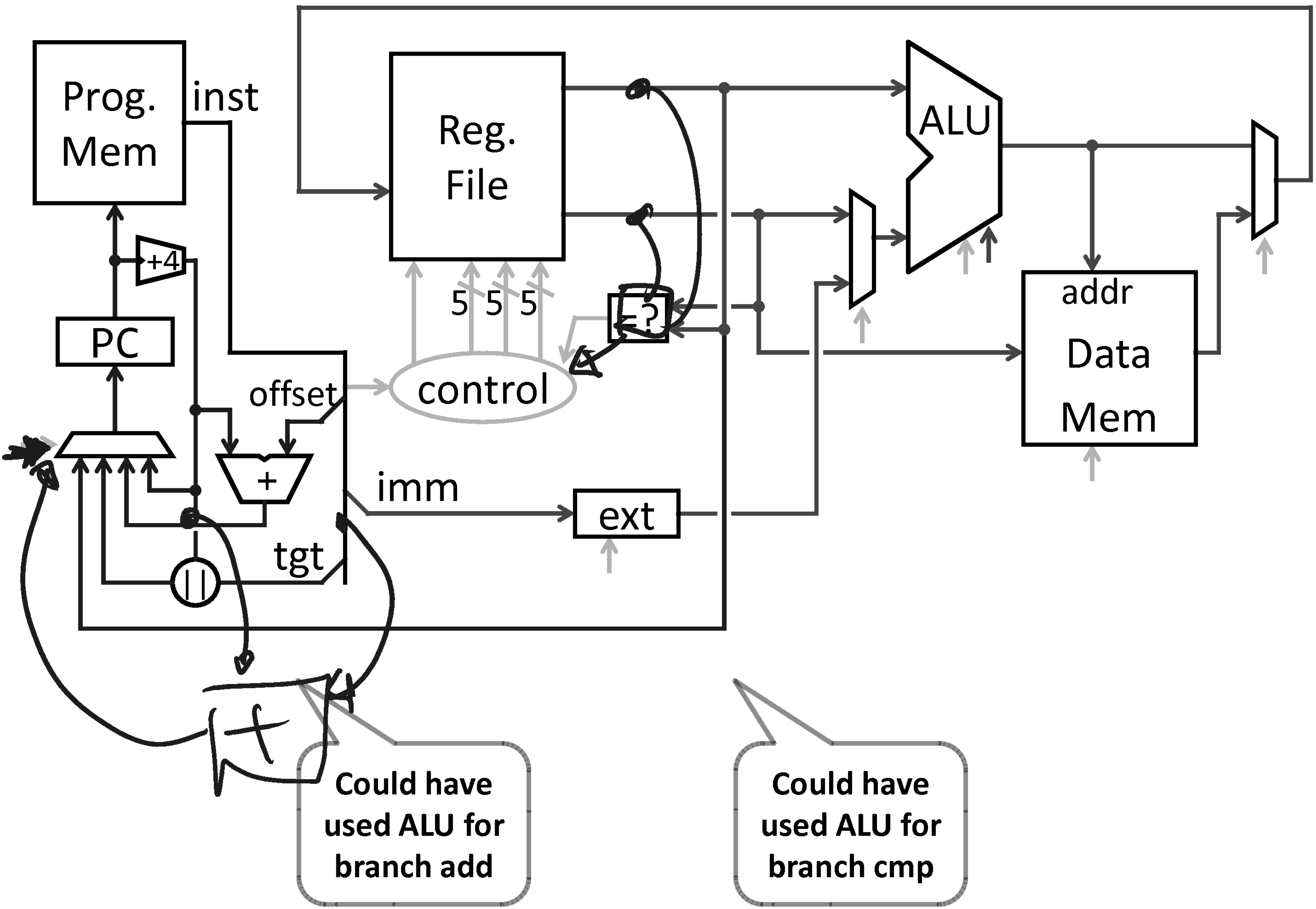
```
if (i == j) { i = i * 4; }
else { j = i - j; }
```

— if

— else

```
BNE r5, r6, ELSE (+8)
    SLL r5, r5, 2
    BEQ r0, r0, BOT (+4)
ELSE: SUB r6, r5, r6
BOT:
```


Prog.
Mem
inst
Reg.
File
ALU
addr
Data
Mem
5
5
5
=?
control
offset
PC
+4
+
imm
ext
tgt
Could have used ALU for branch add
Could have used ALU for branch cmp


Prog. Mem
inst
Reg. File
5 5 5
control
=?
ALU
addr
Data Mem
PC
+4
offset
+
imm
ext
tgt
||
+
Could have used ALU for branch add
Could have used ALU for branch cmp

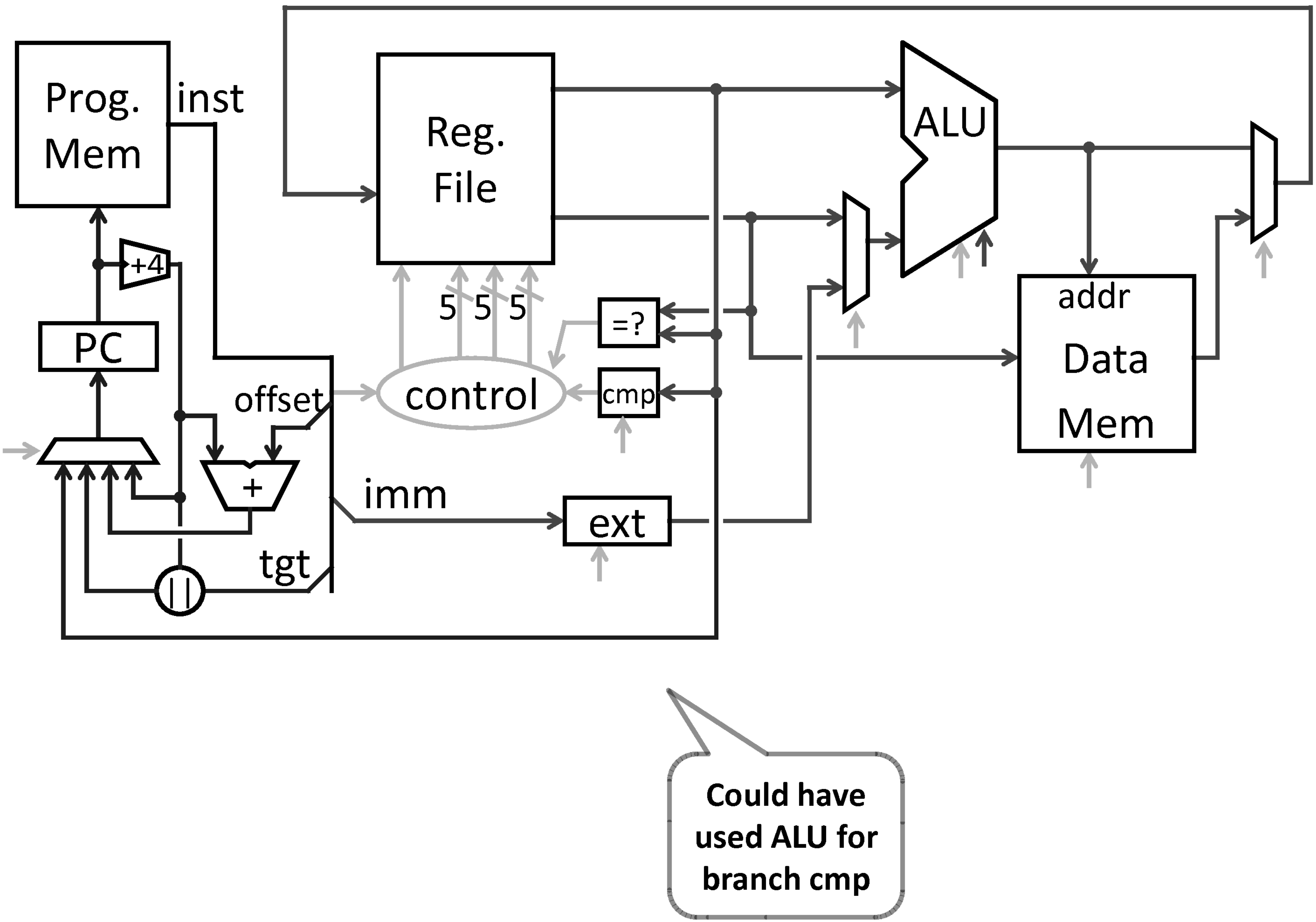
00000100101000010000000000000010

op	rs	subop	offset
6 bits	5 bits	5 bits	16 bits

almost I-Type

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)



Prog. Mem
inst
Reg. File
ALU
+4
PC
5
5
5
=?
cmp
control
offset
+
imm
ext
tgt
||
addr
Data Mem
Could have used ALU for branch cmp

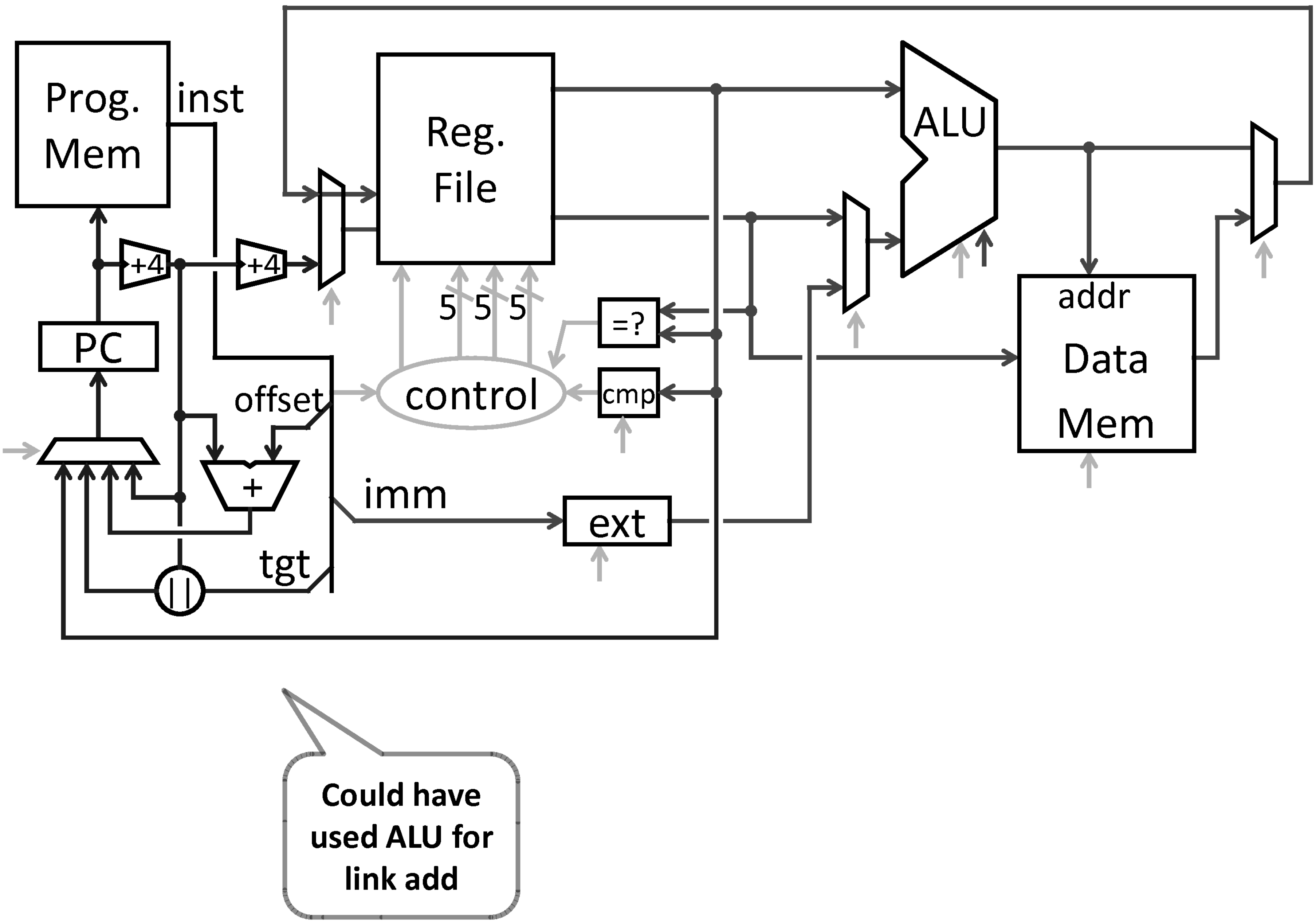
000011 00000001001000011000000010

op	immediate
6 bits	26 bits

J-Type

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

J


Prog.
Mem
inst
Reg.
File
ALU
addr
Data
Mem
PC
+4
+4
5 5 5
=?
cmp
control
offset
imm
ext
+
tgt
||
Could have used ALU for link add