MIPS design

Isaac Pérez Andrade

ITESM Guadalajara
Department of Engineering & Architecture
Department of Computer Science

August - December 2020

A basic Microprocessor

Basic μP

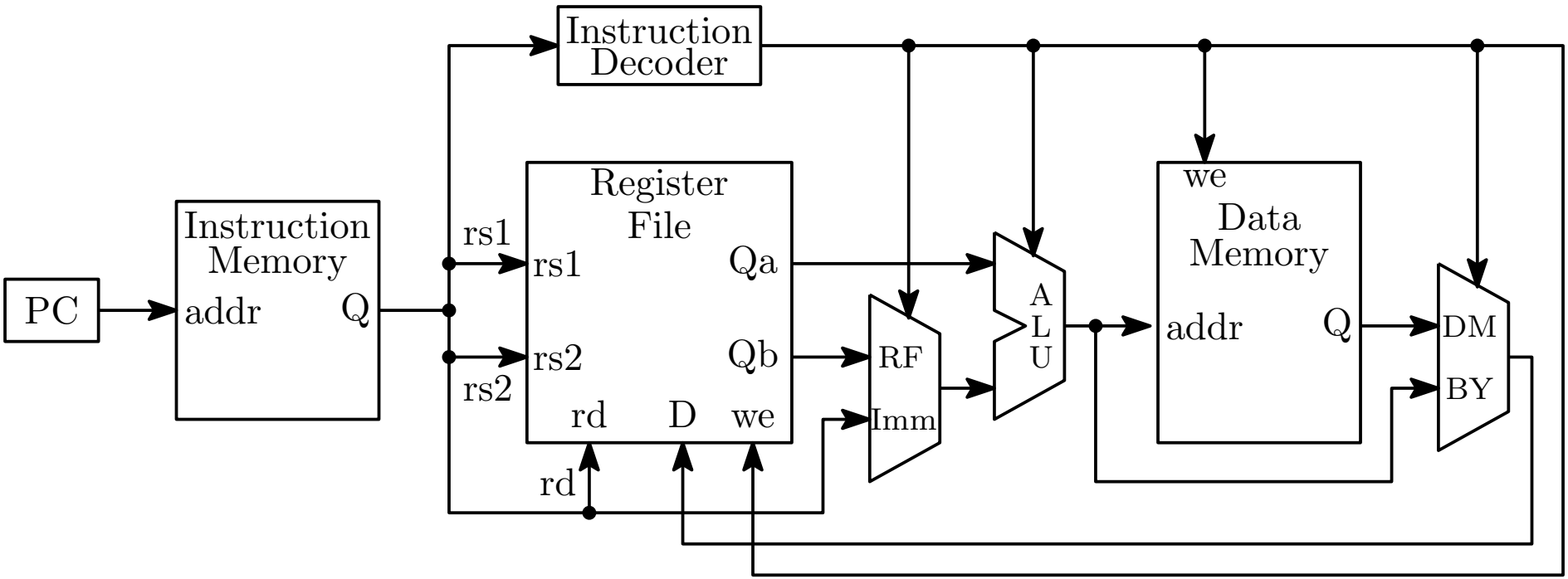


Figure 1: Generalised Microprocessor (μP) schematic.

Basic μP

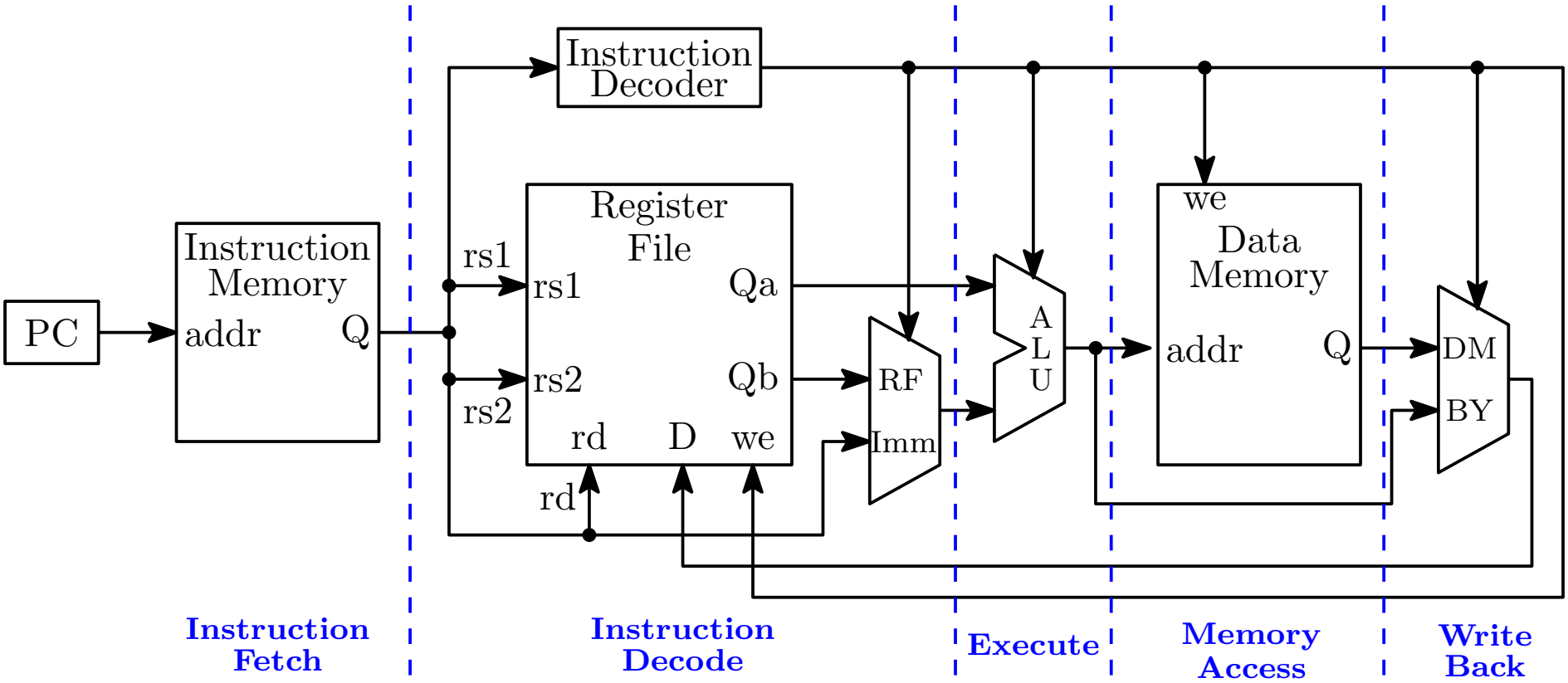


Figure 2: Stages of a μP cycle.

Stages of a basic μP instruction cycle

1. **Instruction Fetch (IF)**. Instructions are read from Instruction Memory (IM).
2. **Instruction Decode (ID)**. Type of operation and operands are defined.
3. **Execute (EXE)**. Operands are used in order to perform arithmetic or logical operations.
4. **Memory Access (MEM)**. Data is read/written from/to Data Memory (DM).
5. **Write Back (WB)**. Results from EXE or MEM stages are written back into Register File (RF).

MIPS example

MIPS

- Microprocessor without Interlocked Pipeline Stage (MIPS) is a Reduced Instruction Set Computer (RISC)-type μP.
- MIPS flavours could use 16, 32- or 64-bits data widths.
- Supports 3 main instruction types.
  - R-Type for register-register operations.
  - I-Type for immediate operations.
  - J-Type for jump operations.

MIPS instruction encoding

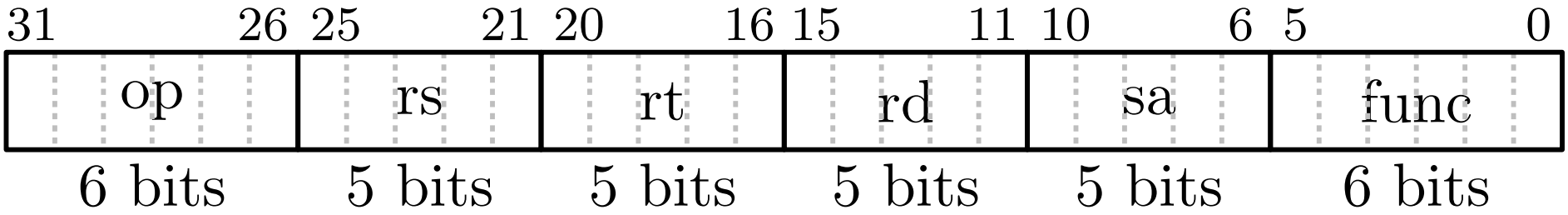


Figure 3: Example of MIPS encoding.

- **op:** Basic operation of the instruction, traditionally called opcode.
- **rs:** First register source operand.
- **rt:** Second register source operand.
- **rd:** Destination register.
- **sa:** Shift amount.
- **func:** Function. Specifies a variant of the operation in the op field.

MIPS instruction encoding

- Considering the previous encoding
  - What type(s) of addressing modes are available?
  - How many registers can we access?
  - Can we access data from the memory?
- Since MIPS is a RISC, we can't simply add bits to the encoding.
  - Why?
- Instead, we must use the same 32-bits for accessing memory.
- For this purpose, some of the fields will have to be modified.

MIPS instruction encoding

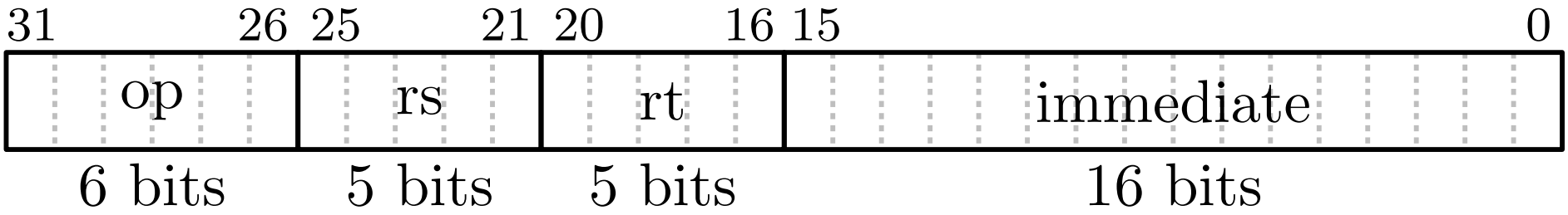


Figure 4: Example of another MIPS encoding.

- This format will allow us to access memory using a 16-bit address.
- This will also allow us to use immediate values (constants) for arithmetic and logical operations.

MIPS instruction encoding

These are the encodings for each of the 3 main MIPS instruction types.

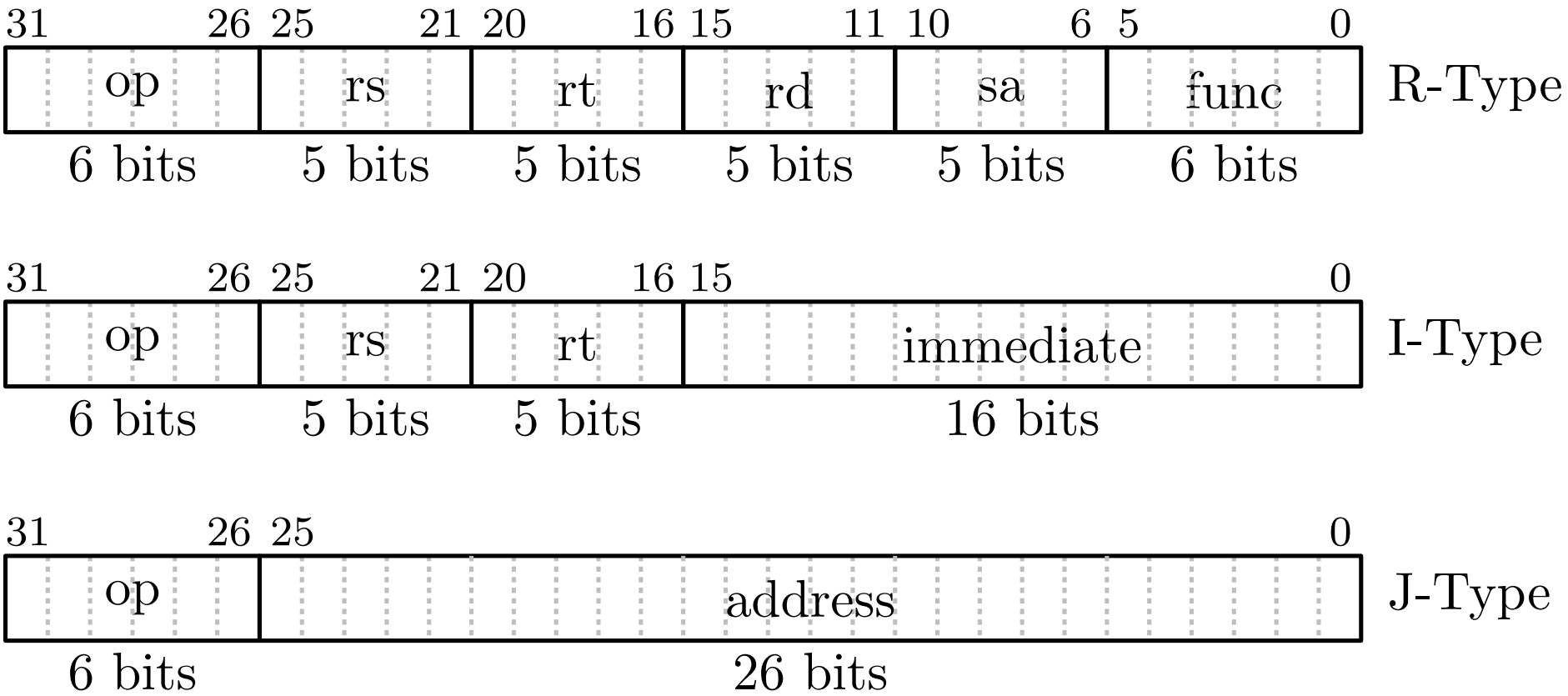


Figure 5: MIPS32 instruction types.

MIPS instruction fields

- **op:** Basic operation of the instruction, traditionally called opcode.
- **rs:** First register source operand.
- **rt:** Second register source operand.
- **rd:** Destination register.
- **sa:** Shift amount.
- **func:** Function. Specifies a variant of the operation in the op field.
- **immediate:** A value that defines an address or a constant.
- **address:** An absolute address value.

Inst.	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	Meaning
add	000000	rs	rt	rd	00000	100000	Register add
sub	000000	rs	rt	rd	00000	100010	Register subtract
and	000000	rs	rt	rd	00000	100100	Register AND
or	000000	rs	rt	rd	00000	100101	Register OR
xor	000000	rs	rt	rd	00000	100110	Register XOR
sll	000000	00000	rt	rd	sa	000000	Shift left
srl	000000	00000	rt	rd	sa	000010	Logical shift right
sra	000000	00000	rt	rd	sa	000011	Arithmetic shift right
jr	000000	rs	00000	00000	00000	001000	Register jump
addi	001000	rs	rt		Immediate		Immediate add
andi	001100	rs	rt		Immediate		Immediate AND
ori	001101	rs	rt		Immediate		Immediate OR
xori	001110	rs	rt		Immediate		Immediate XOR
lw	100011	rs	rt		offset		Load memory word
sw	101011	rs	rt		offset		Store memory word
beq	000100	rs	rt		offset		Branch on equal
bne	000101	rs	rt		offset		Branch on not equal
lui	001111	00000	rt		immediate		Load upper immediate
j	000010			address			Jump
jal	000011			address			Call

Figure 6: MIPS32 integer instructions.

MIPS instruction examples

Table 1: MIPS instruction examples

Instruction	Meaning
`add rd, rs, rt`	`rd ← rs + rt`
`addi rt, rs, imm`	`rt ← rs + (sign_ext)imm`
`ori rt, rs, imm`	`rt ← rs | (zero_ext)imm`
`sra rd, rt, sa`	`rd ← rt « sa`
`lui rt, imm`	`rt ← imm « 16`
`lw rt, offset(rs)`	`rt ← memory[rs + offset]`
`sw rt, offset(rs)`	`memory[rs + offset] ← rt`
`bne rs, rt, label`	`if (rs != rt) then PC ← label`
`j target`	`PC ← target`
`jr rs`	`PC ← rs`

MIPS encoding example

Use MIPS encoding format in order to translate the following assembler code to machine code.

1. `add r4, r5, r6`
2. `sub r4, r5, r6`
3. `subu r4, r5, r6`
4. `addi r7, r8, -10`
5. `addiu r7, r8, -2`
6. `and r7, r8, r6`
7. `andi r7, r8, -1`
8. `lw r12, 100(r4)`
9. `sw r12, 100(r4)`
10. `lw r12, -100(r4)`
11. `sw r12, -100(r4)`
12. `j 1234`
13. `jr r23`

MIPS encoding example

`add r4, r5, r6`

Mnemonic	`add`
Description	Adds two registers and stores result in a register.
Operation	`rd = rs + rt`
Syntax	`add rd, rs, rt`
Encoding	0000 00 ss sss t tttt dddd d 000 00 10 0000

- `add r4, r5, r6` is encoded as

  0000 00 00 101 0 0110 0010 0 000 00 10 0000

  00A62020

MIPS encoding example

`sub r4, r5, r6`

Mnemonic	`sub`
Description	Subtracts two registers and stores result in a register.
Operation	`rd = rs- rt`
Syntax	`sub rd, rs, rt`
Encoding	0000 00 ss sss t tttt dddd d 000 00 10 0010

- `sub r4, r5, r6` is encoded as

  0000 00 00 101 0 0110 0010 0 000 00 10 0010

  00A62022

MIPS encoding example

`subu r4, r5, r6`

Mnemonic	`subu`
Description	Subtracts unsigned registers and stores result in a register.
Operation	`rd = rs- rt`
Syntax	`subu rd, rs, rt`
Encoding	0000 00 ss sss t tttt dddd d 000 00 10 0011

- `subu r4, r5, r6` is encoded as

  0000 00 00 101 0 0110 0010 0 000 00 10 0011

  00A62023

MIPS encoding example

`addi r7, r8, -10`

Mnemonic	`addi`
Description	Signed addition of a register and a sign-extended immediate, stores result in a register.
Operation	`rt = rs + immediate`
Syntax	`addi rt, rs, immediate`
Encoding	0010 00 ss sss t tttt iiii iiii iiii iiii

- `addi r7, r8, -10` is encoded as

  0010 00 00 111 0 1000 1111 1111 1111 0110

  20E8FFF6

MIPS encoding example

`addiu r7, r8, -2`

Mnemonic	`addiu`
Description	Unsigned addition of a register and a sign-extended immediate, stores result in a register.
Operation	`rt = rs + immediate`
Syntax	`addiu rt, rs, immediate`
Encoding	0010 01 ss sss t tttt iiii iiii iiii iiii

- `addiu r7, r8, -2` is encoded as

  0010 01 00 111 0 1000 1111 1111 1111 1110

  28E8FFFE

MIPS encoding example

`and r7, r8, r6`

Mnemonic	`and`
Description	Bitwise `AND` between two registers, stores result in a registe
Operation	`rd = rs & rt`
Syntax	`and rd, rs, rt`
Encoding	0000 00 ss sss t tttt dddd d 000 00 10 0100

- `and r7, r8, r6` is encoded as

  0000 00 01 000 0 0110 0011 1 000 00 10 0100

  01063824

MIPS encoding example

`andi r7, r8, -1`

Mnemonic	`andi`
Description	Bitwise `AND` between a register and a zero-extended immediate, stores result in register.
Operation	`rt = rs & immediate`
Syntax	`andi rt, rs, immediate`
Encoding	0011 00 ss sss t tttt iiii iiii iiii iiii

- `andi r7, r8, -1` is encoded as

  0011 00 01 000 0 0111 0000 0000 1111 1111

  310700FF

MIPS encoding example

`lw r12, 100(r4)`

Mnemonic	`lw`
Description	A word is loaded from memory into a register
Operation	`rt = Mem[offset + rs ]`
Syntax	`lw rt, offset (rs)`
Encoding	1000 11 ss sss t tttt iiii iiii iiii iiii

- `lw r12, 100(r4)` is encoded as

  1000 11 00 100 0 1100 0000 0000 0110 0100

  8C8C0064

MIPS encoding example

`sw r12, 100(r4)`

Mnemonic	`sw`
Description	A word is stored from a register into memory
Operation	`Mem[offset + rs ] = rt`
Syntax	`sw rt, offset (rs)`
Encoding	**1010 11 ss sss t tttt iiii iiii iiii iiii**

- `sw r12, 100(r4)` is encoded as

  1010 11 00 100 0 1100 0000 0000 0110 0100

  AC8C0064

MIPS encoding example

`lw r12, -100(r4)`

Mnemonic	`lw`
Description	A word is loaded from memory into a register
Operation	`rt = Mem[offset + rs ]`
Syntax	`lw rt, offset (rs)`
Encoding	1000 11 ss sss t tttt iiii iiii iiii iiii

- `lw r12, -100(r4)` is encoded as

  1000 11 00 100 0 1100 1111 1111 1001 1100

  8C8CFF9C

MIPS encoding example

`sw r12, -100(r4)`

Mnemonic	`sw`
Description	A word is stored from a register into memory
Operation	`Mem[offset + rs ] = rt`
Syntax	`sw rt, offset (rs)`
Encoding	**1010 11 ss sss t tttt iiii iiii iiii iiii**

- `sw r12, -100(r4)` is encoded as

  1010 11 00 100 0 1100 1111 1111 1001 1100

  AC8CFF9C

MIPS encoding example

`j 1234`

Mnemonic	`j`
Description	Jump to address
Operation	`PC` ← **`address`**
Syntax	`j` **`address`**
Encoding	**0000 10** **ii iiii iiii iiii iiii iiii iiii**

- `j 1234` is encoded as

  0000 10 **00 0000 0000 0000 0100 1101 0010**

  080004D2

MIPS encoding example

`jr r23`

Mnemonic	`jr`
Description	Jump to address in register
Operation	`PC` ← `rs`
Syntax	`jr rs`
Encoding	**0000 00 ss sss 0 0000 0000 0000 0000 1000**

- `jr r23` is encoded as

  0000 00 10 111 0 0000 0000 0000 0000 1000

Custom MIPS design

Custom MIPS design

- We will begin to design the Microarchitecture (μA) for each stage of a custom MIPS μP.
- We will use an incremental approach, *i.e.*, basic features will be improved over time.
- We will focus on single-cycle design at first.

Custom MIPS design

- We will divide our MIPS into R-, I- and J-Type instructions.
- We will start by implementing R-Type instructions.
- I- and J-Type instructions will be added incrementally.
- We will analyse the necessary datapath for implementing each instruction type.

Custom MIPS design

General design specifications

- Instruction set based (but modified) on 16-bit MIPS μP.
- Single-cycle design.
- Data width is 16-bits.
- Instruction encoding (instruction width) is 32-bits long.
- RF contains 32 registers.
  - `r0` **must** always be 0, *i.e.*, `r0` is not writeable. Useful for `load` and `store` operations.
  - `r31` is return address. More about this later.

PC datapath

PC datapath

- Program Counter (PC) indicates the instruction being executed.
- PC is effectively the read address of IM.
- PC **updates** its value every clock cycle.

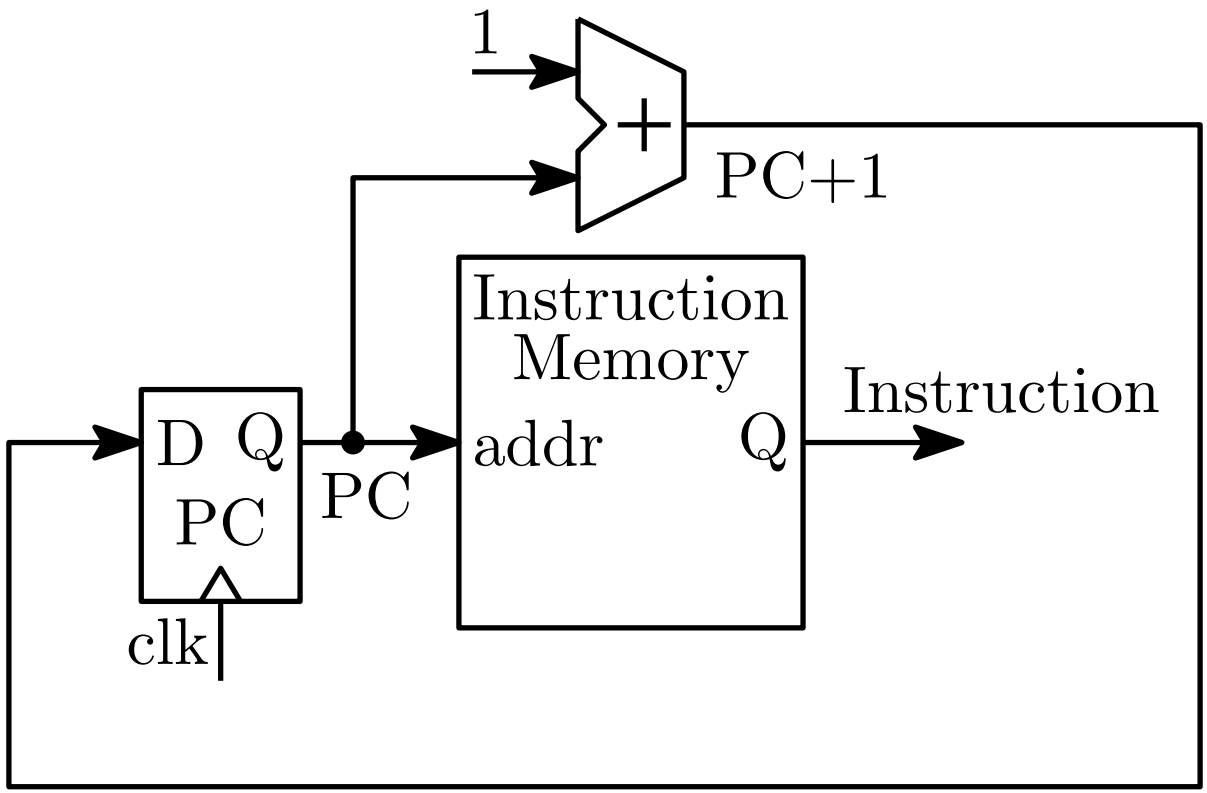


Figure 7: Basic PC datapath.

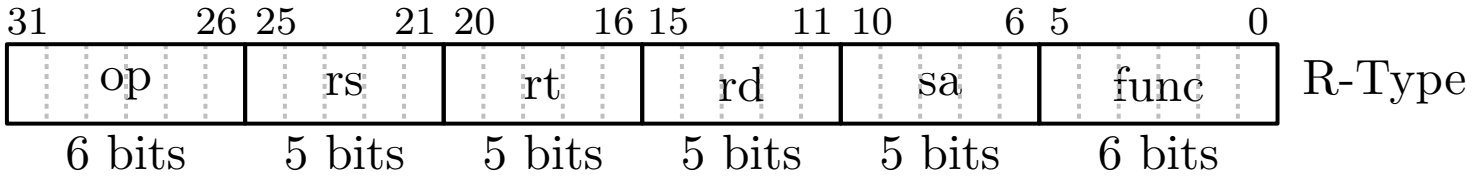
R-Type datapath

Custom MIPS design

- R-Type instructions are instructions in which ALU operands are retrieved from RF and result is stored back in RF as well.

Instruction	Syntax	Meaning
ADD	ADD rd, rs, rt	Reg[rd] ← Reg[rs] + Reg[rt]
SUB	SUB rd, rs, rt	Reg[rd] ← Reg[rs] - Reg[rt]
NAND	NAND rd, rs, rt	Reg[rd] ← ∼(Reg[rs] & Reg[rt])
NOR	NOR rd, rs, rt	Reg[rd] ← ∼(Reg[rs] | Reg[rt])
XNOR	XNOR rd, rs, rt	Reg[rd] ← ∼(Reg[rs] ^ Reg[rt])
AND	AND rd, rs, rt	Reg[rd] ← Reg[rs] & Reg[rt]
OR	OR rd, rs, rt	Reg[rd] ← Reg[rs] | Reg[rt]
XOR	XOR rd, rs, rt	Reg[rd] ← Reg[rs] ^ Reg[rt]
SLL	SLL rd, rs, sa	Reg[rd] ← Reg[rs] « sa
SRL	SRL rd, rs, sa	Reg[rd] ← Reg[rs] » sa
SLA	SLA rd, rs, sa	Reg[rd] ← Reg[rs] «< sa
SRA	SRA rd, rs, sa	Reg[rd] ← Reg[rs] »> sa

R-Type arithmetic/logic instruction




```
sub rd, rs, rt
```

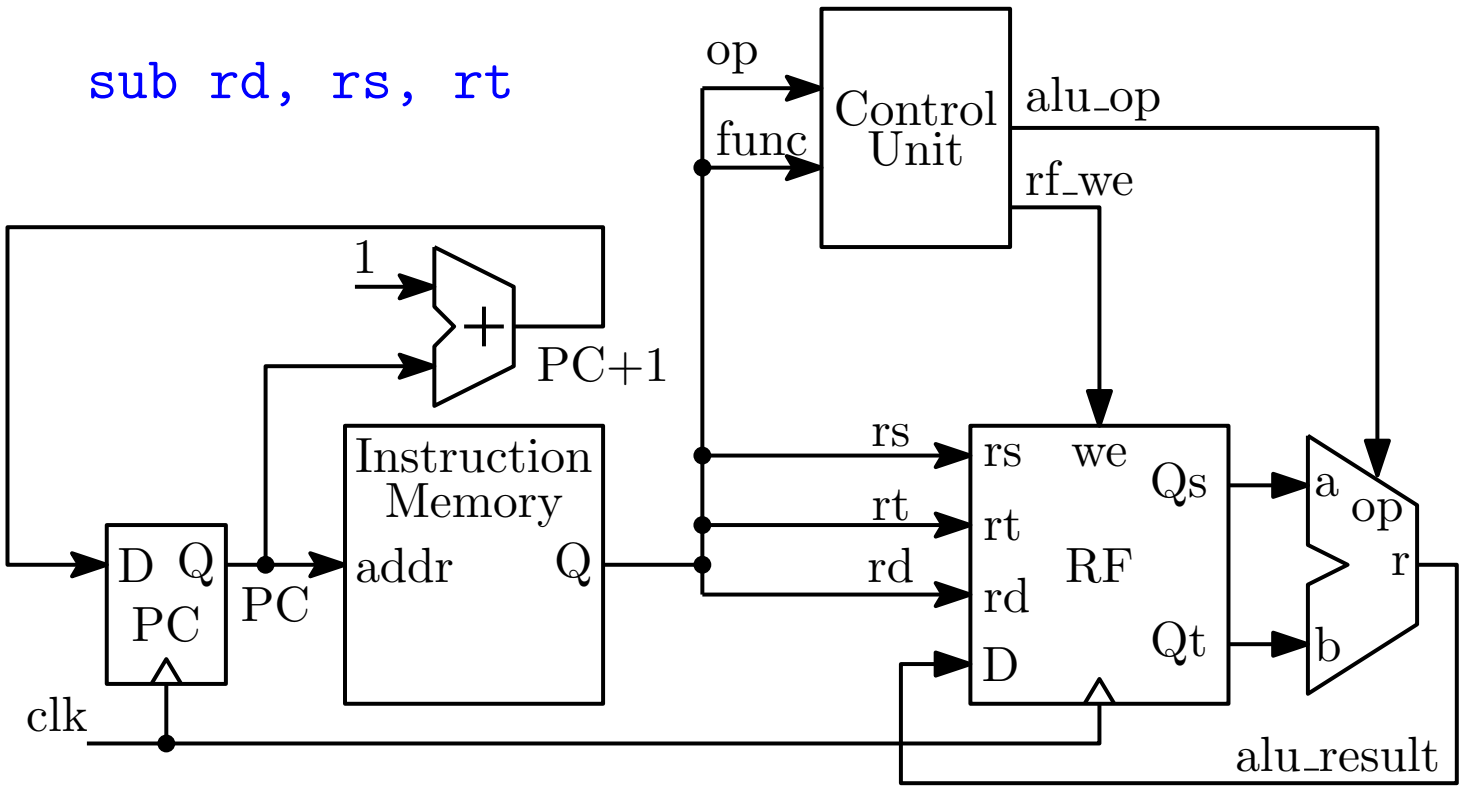


Figure 8: μA of MIPS R-Type arithmetic/logic instructions datapath.

R-Type arithmetic/logic instructions datapath

- Arithmetic and Logic Unit (ALU) takes its operands `a` and `b` from outputs `Qs` and `Qt` from RF, respectively.
- Control unit asserts `alu_op` according to value of `func` in order to instruct ALU the type of operation to be performed, *i.e.*, `add`, `sub`, `and`, *etc.*
- Control unit asserts `rf_we` in order to write the `alu_result` back into RF.

R-Type shift instructions datapath

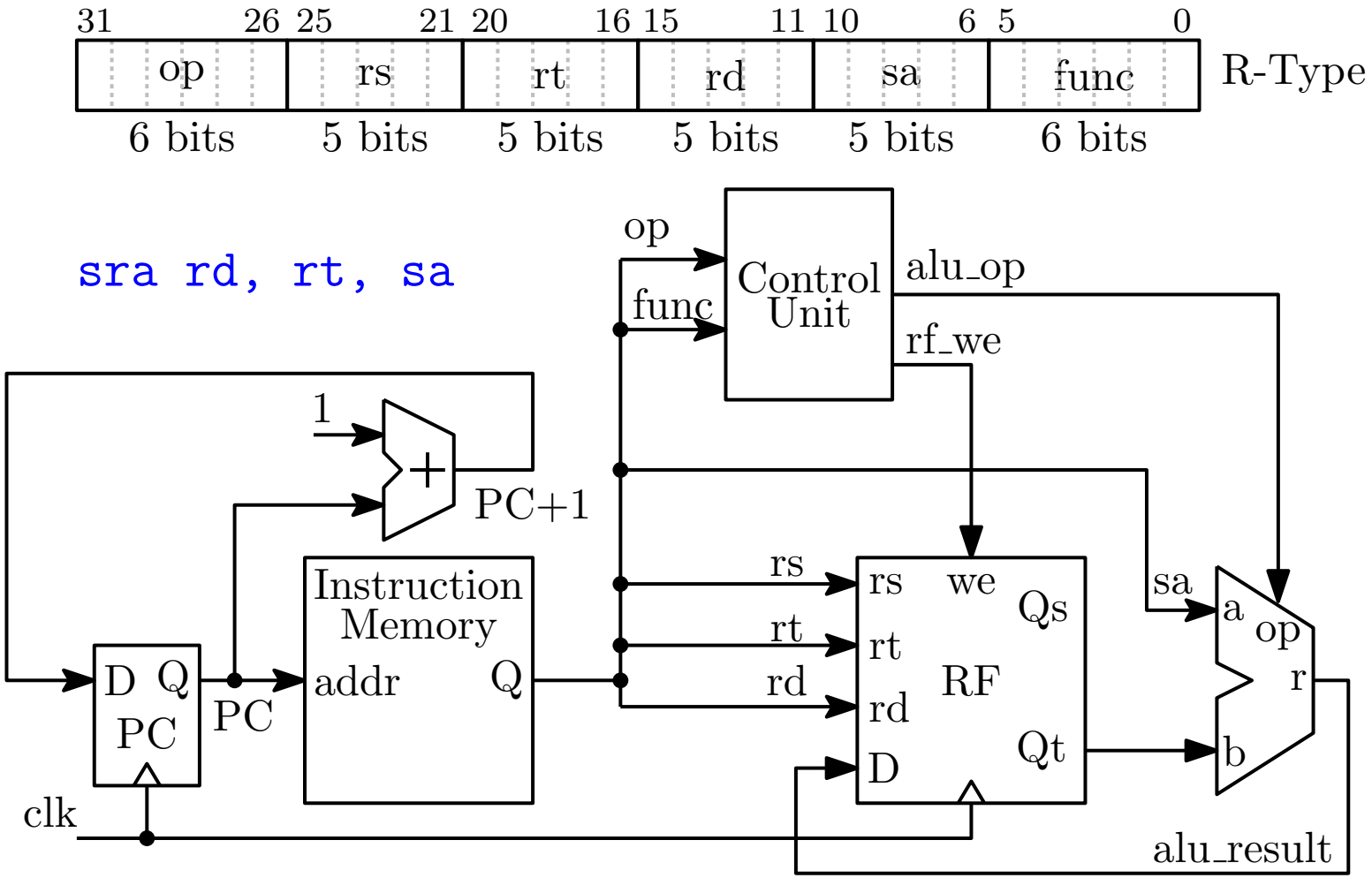


Figure 9: μA of MIPS R-Type shift instructions datapath.

R-Type shift instructions

- Input `a` in ALU represents shift amount `sa`.
- `sa` must be zero-extended.
- `rs` and `Qs` are not used.

R-Type instructions datapath

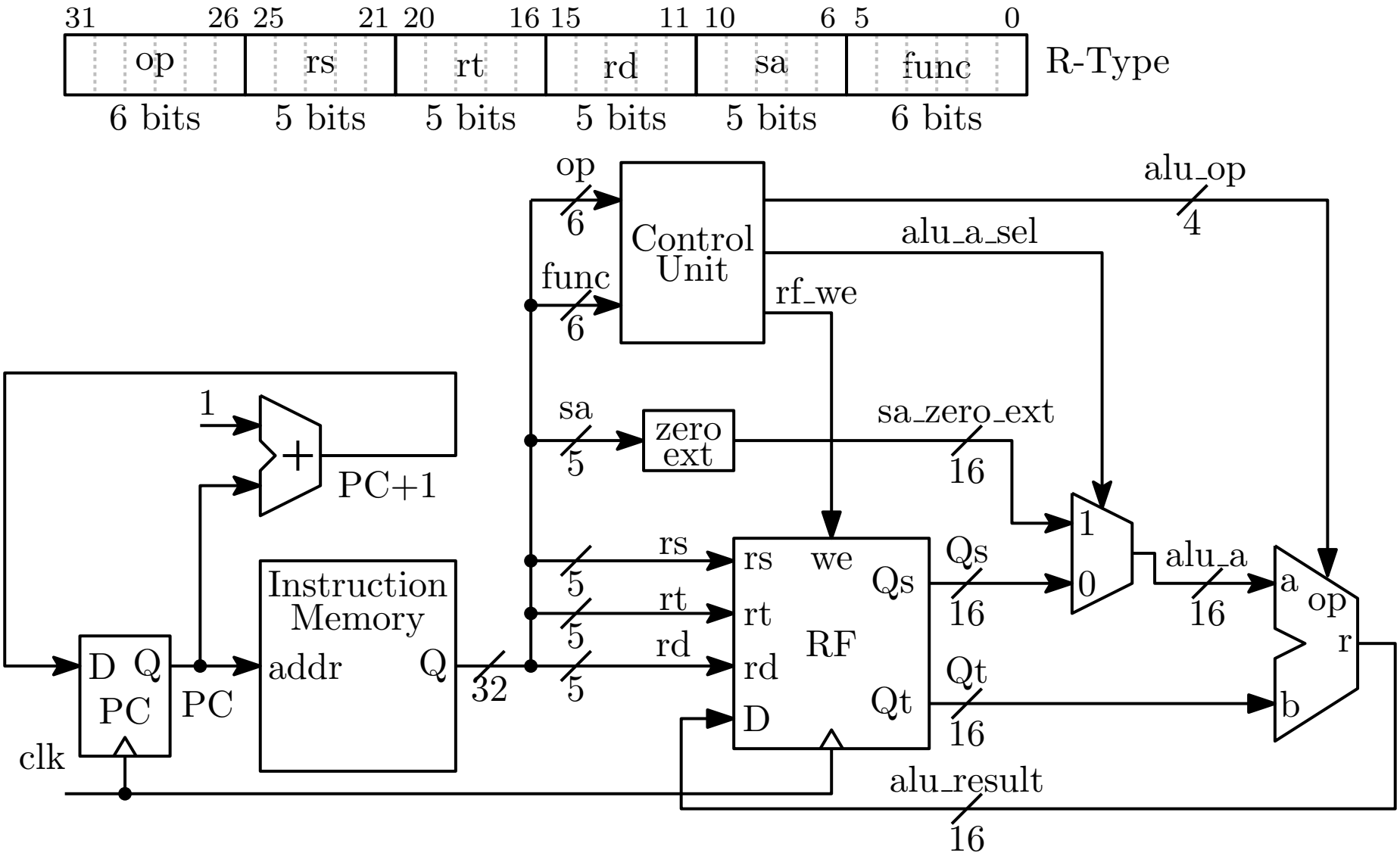


Figure 10: μA of MIPS R-Type instructions datapath.

I-Type datapath

I-Type arithmetic/logic instructions datapath

- I-Type instructions use immediate addressing mode.

Instruction	Syntax	Meaning
ADDI	ADDI rt, rs, imm	Reg[rt] ← Reg[rs] + imm
SUBI	SUBI rt, rs, imm	Reg[rt] ← Reg[rs] - imm
ANDI	ANDI rt, rs, imm	Reg[rt] ← Reg[rs] & imm
ORI	ORI rt, rs, imm	Reg[rt] ← Reg[rs] | imm
XORI	XORI rt, rs, imm	Reg[rt] ← Reg[rs] ^ imm
LUI	LUI rt, imm	Reg[rt] ← {imm[7:0], 8'b0}
LLI	LLI rt, imm	Reg[rt] ← {8'b0, imm[7:0]}
LW	LW rt, imm(rs)	Reg[rt] ← Mem[rs+imm]
SW	SWR rt, imm(rs)	Mem[rs+imm] ← Reg[rt]
BEQ	BEQ rt, rs, imm	if (Reg[rt] == Reg[rs]) then PC ← imm
BNE	BNE rt, rs, imm	if (Reg[rt] != Reg[rs]) then PC ← imm

I-Type arithmetic/logic instructions datapath

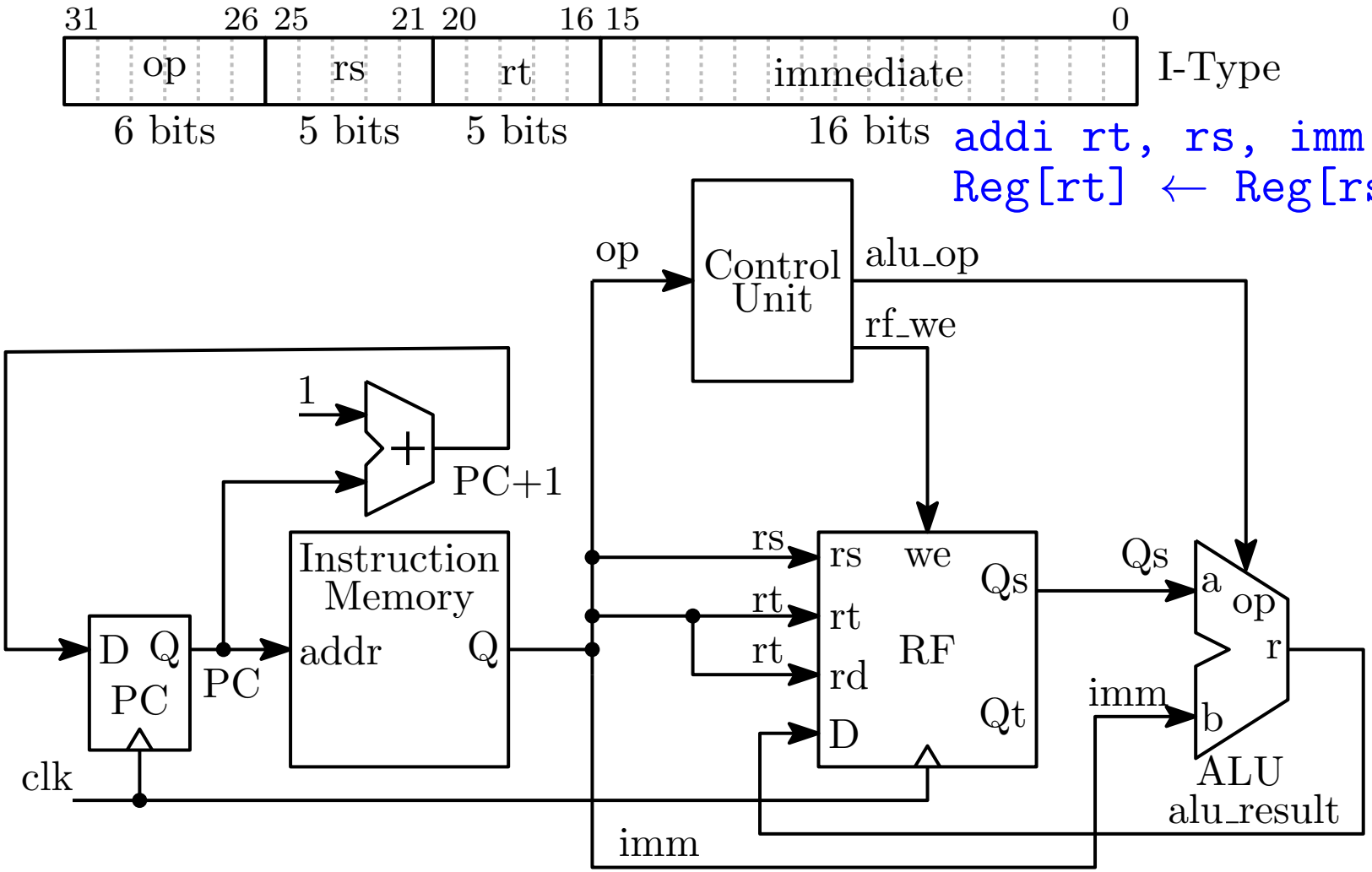


Figure 11: μA of a MIPS I-Type arithmetic/logic instruction.

I-Type load instructions datapath

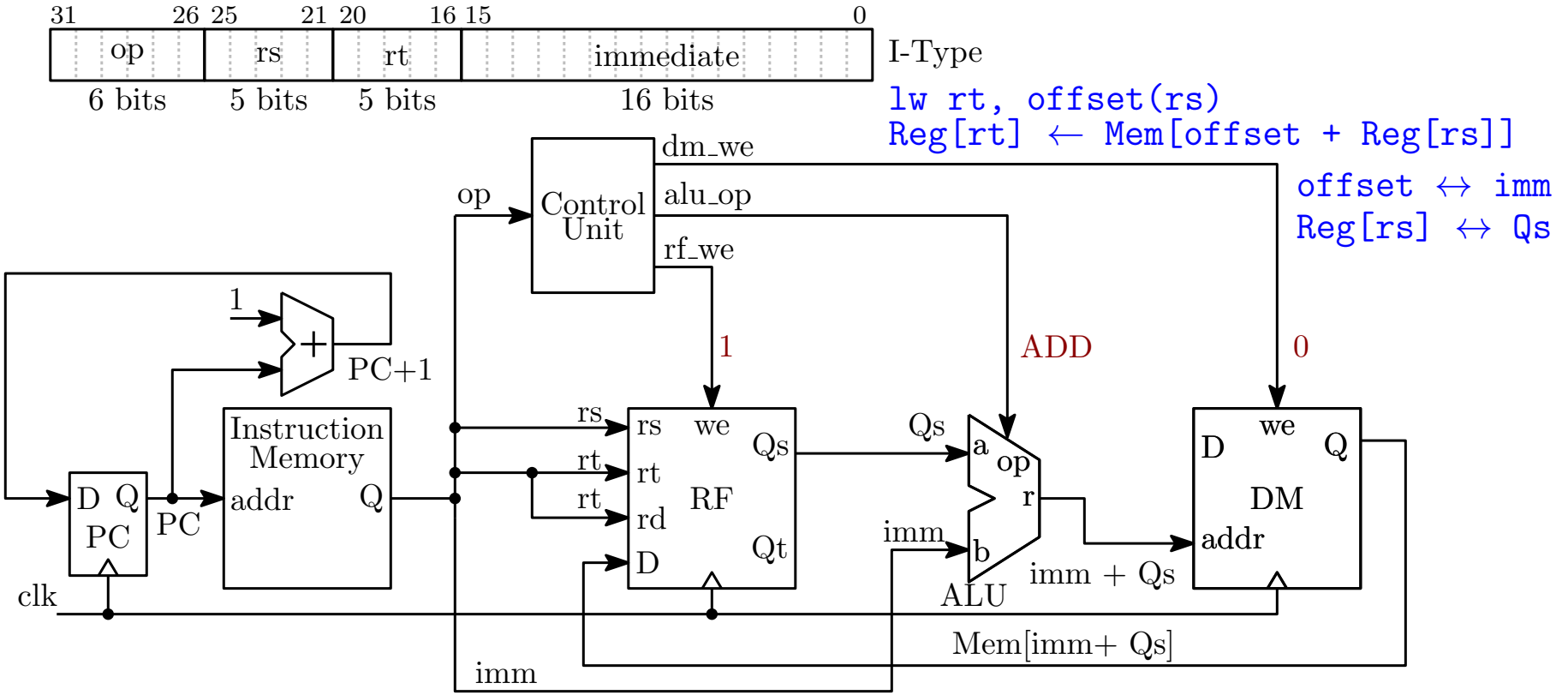


Figure 12: μA of a MIPS I-Type load instruction datapath.

I-Type store instructions datapath

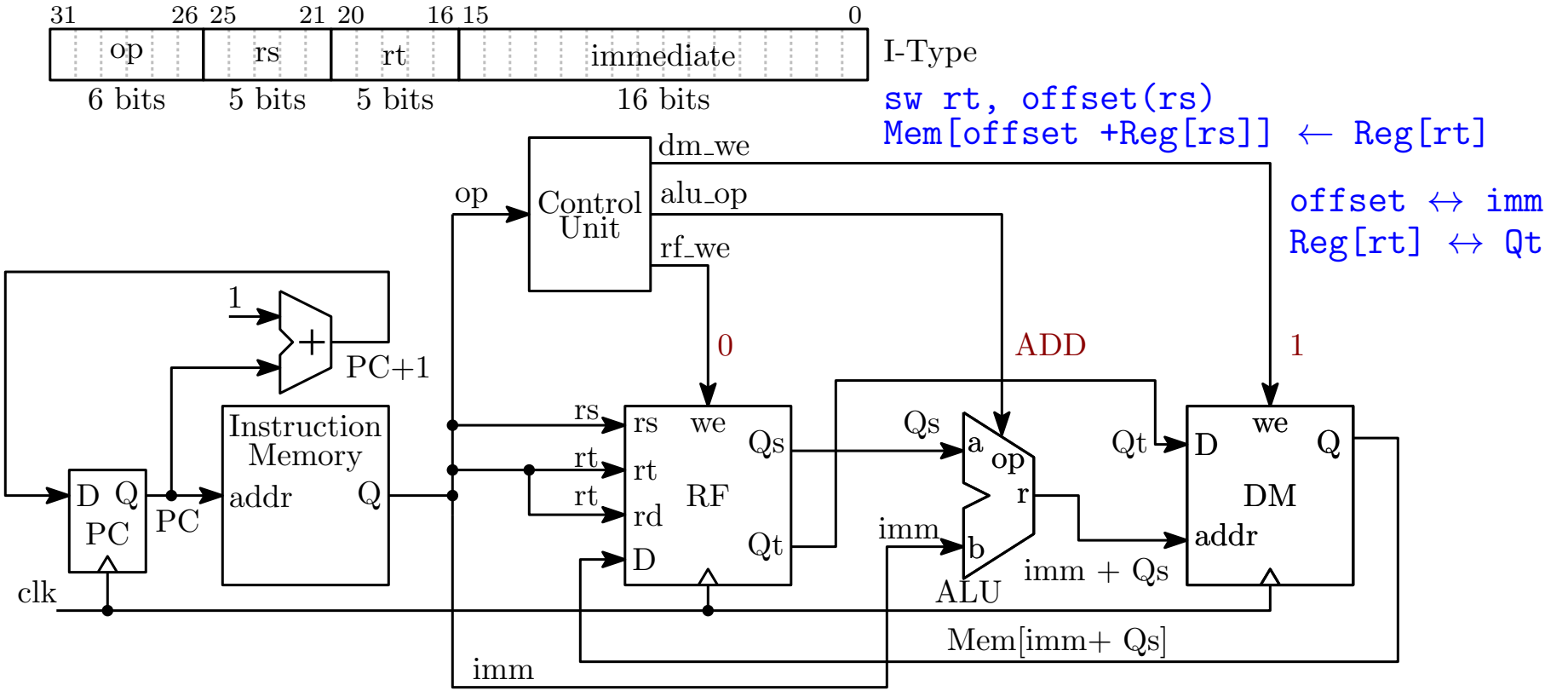


Figure 13: μA of a MIPS I-Type store instruction datapath.

I-Type load/store instructions

- ALU will perform calculations in order to find the displacement address to read/write data from/to.
  - `lw rt, offset(rs)` is equivalent to

    `Reg[rt]` $\leftarrow$ `Mem[offset+Reg[rs]]`

    `sw rt, offset(rs)` is equivalent to

    `Mem[offset+Reg[rs]]` $\leftarrow$ `Reg[rt]`
- Control unit provides a `dm_we` signal, which is a write enable for reading/writing from DM.

I-Type conditional branch instructions

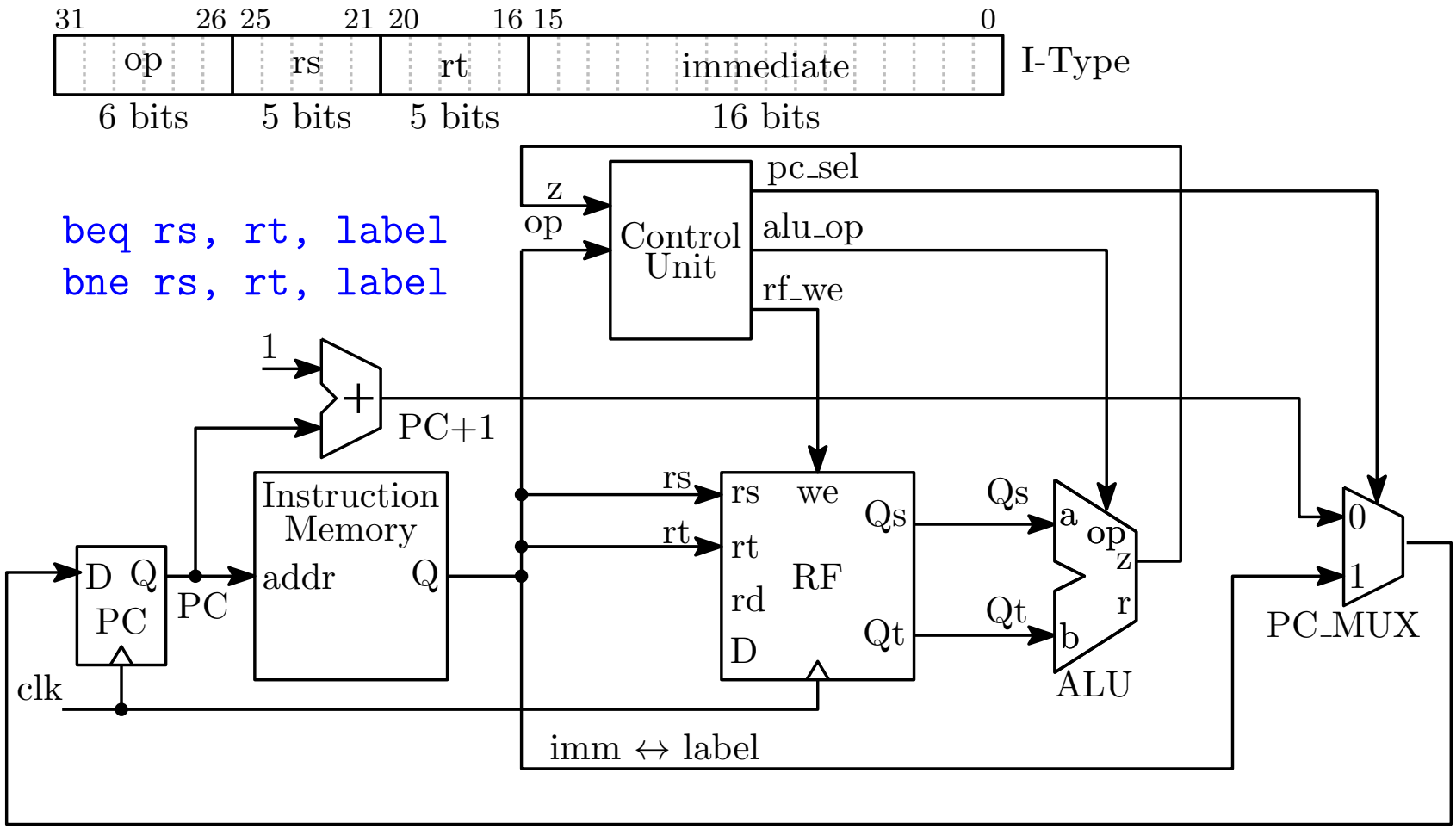


Figure 14: μA of a MIPS I-Type conditional branch instruction.

I-Type load/store instructions

- Contents of `rs` and `rt` are compared in order to determine whether to branch or not.
- If the branch is taken, PC is updated with an `imm` value, which is the target address.
- There is no write-back to RF.

J-Type datapath

J-Type instructions

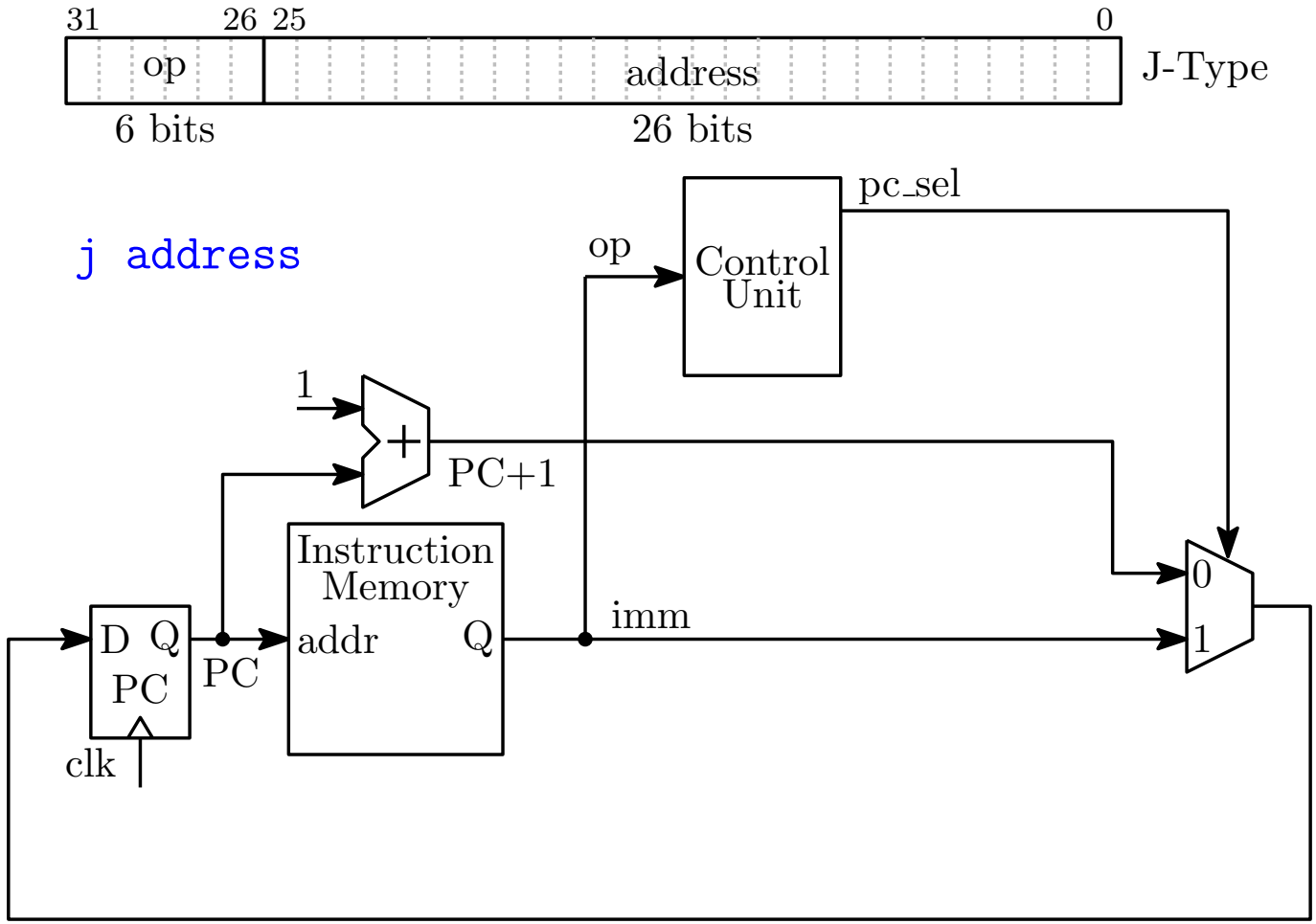


Figure 15: μA of a MIPS J-Type jump instruction.