R-R Architectures (RISC)
DLX ISA

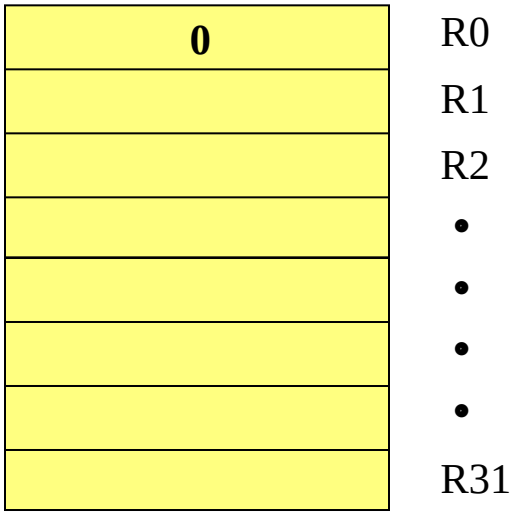
DLX Execution environment

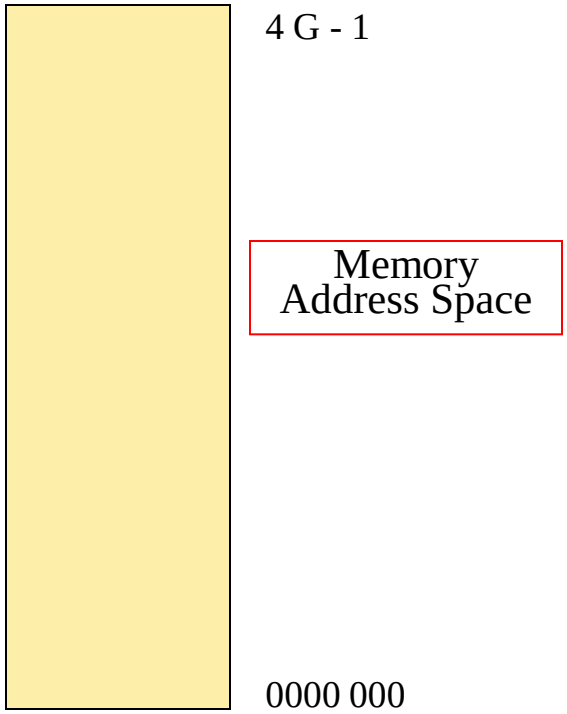
•ORTHOGONALITY BETWEEN REGISTERS AND INSTRUCTIONS (only exception… R31 !)

Program Counter

•Instructions are aligned to addresses multiples of 4
•Stack is handled by software
•In JL-type instructions, the return address is saved onto R31
•There are no FLAGS; conditions are set explicitly into registers by means SET CONDITION instructions:
Scn Rd, Rs1, Rs2

32 general-purpose registers





• I/O: Memory Mapped

Main characteristics of DLX ISA

- 32 32-bit registers for general use GPR (R0..R31, with R0=0).
- 32 32-bit floating-point registers (IEEE 754) (single-precision or double-precision using paired registers); one status register (FLAG) for floating point operations.
- 3 instruction formats (I,J,R).
- Memory address space: 32-bit addresses ➔ 4 GB
- I/0 devices are memory-mapped
- Only one addressing mode: register + offset (16 bits)
- Minimum memory element that can be addressed: byte
- Data and instructions are aligned.
- There are no specific instructions to handle the stack. Stack is handled by software.
- FLAGS generated by the ALU are not accessible by SW through a status register. With the instructions performing a comparison between two operands (Set Condition), the result of the comparison (true/false) is explicitly written into the destination register.
- Interrupts (IAR, only one routine at 0).

The DLX instruction set

- Main **arithmetical and logical (ALU)** instructions also with immediate (I) operand:
  - Logical Instructions: **AND(I), OR(I), XOR(I)**
  - Arithmetical Instructions: **ADD(I), SUB(I), MULT, DIV**
  - Logical shift Instructions (to the right also arithmetical): **SLL(I), SRL(I), SRA(I)**
  - SET CONDITION Instructions: **Scn, ScnI, with cn = EQ, NE, LT, GT, LE, GE**
- Main instructions for **data transfer**
  - Load byte, Load Halfword, Load Word (**LB, LH, LW, LBU, LHU**)
  - Store byte, Store Halfword, Store Word (**SB, SH, SW**)
  - Load/Store Floating Point with single or double precision (**LF/SF and LD/SD**)
  - Copy of a datum from a GPR to a FPR and viceversa (**MOVI2FP and MOVFP2I**)
- Main instructions for **control transfer**
  - Conditional Jump Instructions: **BNEZ, BEQZ**
  - Unconditional Jump Instructions (direct and indirect): **J, JR**
  - Procedure Call Instructions: **JAL, JALR**
  - Return Instruction from the Interrupt service procedure: **RFE**

ALU INSTRUCTIONS

- **Instructions with 3 operands**
  - **2 "*source*" operands (register, register/16-bit immediate operand)**
  - **1 "*destination*" operand (register)**

ADD R1, R2, R3 R1←R2+R3

ADDI R1, R2, 3 R1←R2+3

ADDI R1, **R0**, 5 R1← 0+5 (i.e.: R1← 5)

ADD R1, R5, **R0** R1← R5+R0 (i.e.: R1←R5)

- **ADDU, ADDUI, SUBU, SUBUI….** ***ALU instructions on "unsigned" operands***
- ***"signed" operands: "trap on overflow"***

OTHER ALU INSTRUCTIONS: Set Condition (1)

Comparison instructions belong to the "ALU" class too. These instructions compare two *source* operands and set the *destination* operand to "1" or "0" if, respectively, the comparison condition is satisfied or not.

- "SET EQUAL" (SEQ, =), "SET NOT EQUAL" (SNE, ≠), "SET LESS THAN" (SLT, <), ……..

EXAMPLES:

SLT R1,R2,R3 if (R2<R3): R1 ←1 else R1←0

- In case of immediate source operand:

  SLTI R1,R2,5 if (R2<5): R1 ←1 else R1←0

- In case of comparison between "unsigned" operands:

  SLTU R1,R2,R4 if (R2<R4): R1←1 else R1←0

OTHER ALU INSTRUCTIONS: Set Condition (2)

If suffix *U* is used, operands are seen as "unsigned", viceversa they are seen as integers ("signed") represented in 2's complement.

Instructions	Meaning
S**LT*U*** R1,R2,R3	If (R2<R3) then R1 ← 1 else R1 ← 0
S**GT*U*** R1,R2,R3	If (R2>R3) then R1 ← 1 else R1 ← 0
S**LE*U*** R1,R2,R3	If (R2≤R3) then R1 ← 1 else R1 ← 0
S**GE*U*** R1,R2,R3	If (R2≥R3) then R1 ← 1 else R1 ← 0
S**EQ** R1,R2,R3	If (R2=R3) then R1 ← 1 else R1 ← 0
S**NE** R1,R2,R3	If (R2≠R3) then R1 ← 1 else R1 ← 0
S**LT*U*I** R1,R2,N	If (R2<N) then R1 ← 1 else R1 ← 0
S**GT*U*I** R1,R2,N	If (R2>N) then R1 ← 1 else R1 ← 0
S**LE*U*I** R1,R2,N	If (R2≤N) then R1 ← 1 else R1 ← 0
S**GE*U*I** R1,R2,N	If (R2≥N) then R1 ← 1 else R1 ← 0
S**EQI** R1,R2,N	If (R2=N) then R1 ← 1 else R1 ← 0
S**NEI** R1,R2,N	If (R2≠N) then R1 ← 1 else R1 ← 0

DATA TRANSFER INSTRUCTIONS

- Instructions that access memory (load and store)
- The memory address of the operand is the sum between the content of a register (encoded into RS1) and a 16 bit "offset"
- The instruction is encoded in I-format.
- Examples:

LW R1, 40(R3) $R1 \leftarrow_{32} M[40+R3]$

LB R1, 40(R3) $R1 \leftarrow (M[40+R3]_7)^{24} \,\#\#\, M[40+R3]$

LBU R1,40(R3) $R1 \leftarrow (0)^{24} \,\#\#\, M[40+R3]$

"Sign Extension"

SW 10(R5), R7 $M[10+R5] \leftarrow_{32} R7$

CONTROL TRANSFER INSTRUCTIONS

- "JUMP": unconditional control transfer (or jump)
- "JUMP AND LINK: control transfer which saves the return address (procedure call)
- "BRANCH": conditional control transfer (conditional jump or branch)

"JUMP"

- J alfa PC ← PC + OFFSET(alfa) J-type instructions
  ("PC relative", OFFSET di 26 bit)
- JR R3 PC←R3 I-type instructions

"JUMP AND LINK"

- JAL alfa R31←PC ; PC ← PC + OFFSET(alfa)
- JALR R3 R31←PC ; PC←R3
- JR R31 PC←R31 This is the return instruction from a procedure
  (without a stack only a single nesting level is allowed)

HOW TO SUPPORT NESTED CALLS?

CONTROL TRANSFER INSTRUCTIONS (BRANCHES)

"BRANCH" (I-type instructions)

- "BRANCH EQUAL ZERO": **BEQZ**
- "BRANCH NOT EQUAL ZERO": **BNEZ**

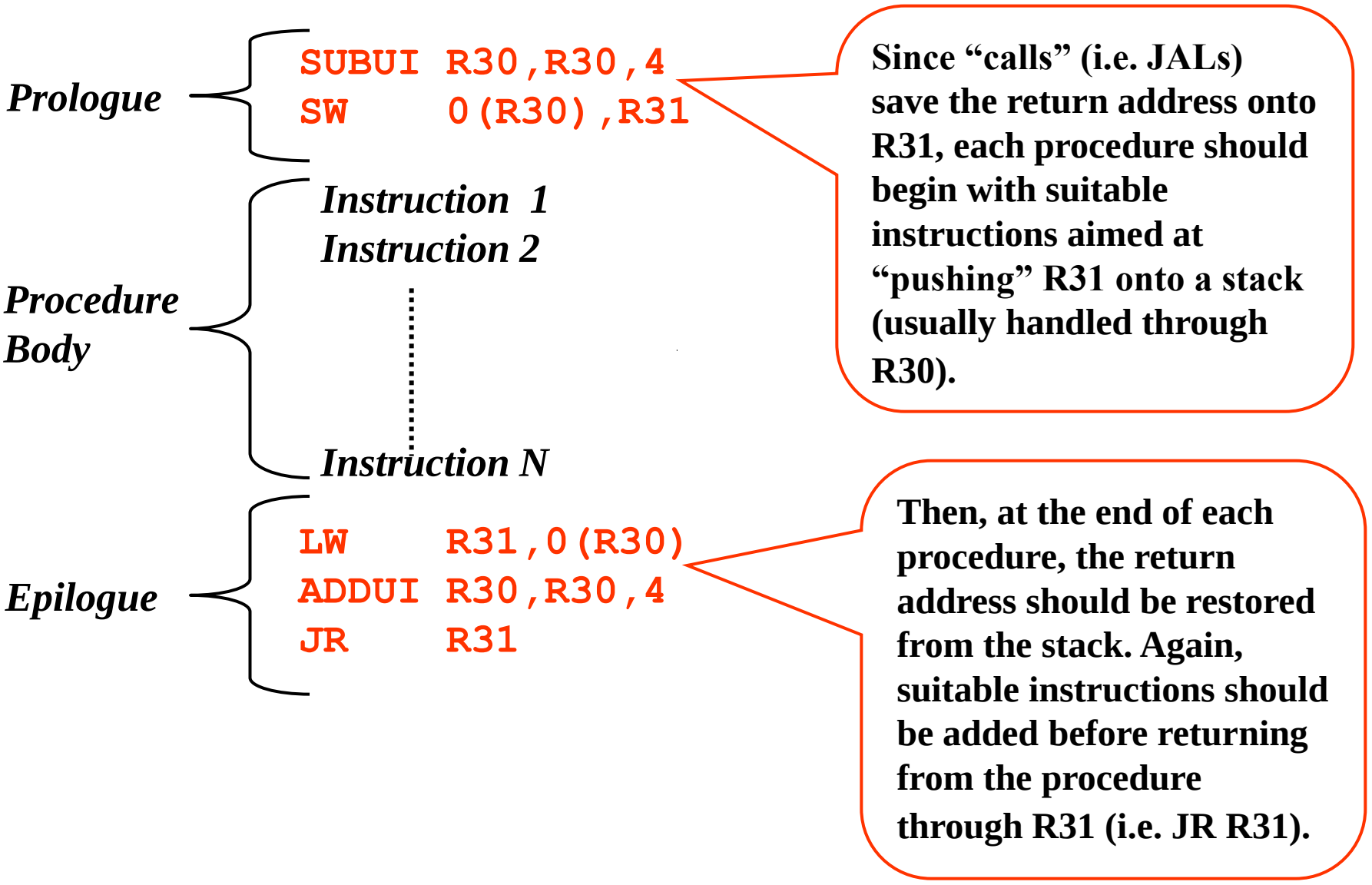
BEQZ R4, alfa
if (R4=0) ➔ PC ← PC + OFFSET(alfa)
BNEZ R4, alfa
if (R4≠0) ➔ PC ← PC + OFFSET(alfa)

("PC relative", 16-bit OFFSET)

With a "set condition" instruction followed by a conditional jump instruction a **Compare and Branch** operation can be carried out (comparison and jump conditioned by the result of the comparison) without the need for any dedicated flag

Example: write the DLX Assembly code that executes the operation C=|A|+B, with A,B,C being variables in memory (32 bit)

How nested calls can be supported ?



Instructions for handling interrupts

- Return from an interrupt handling procedure: **RFE**

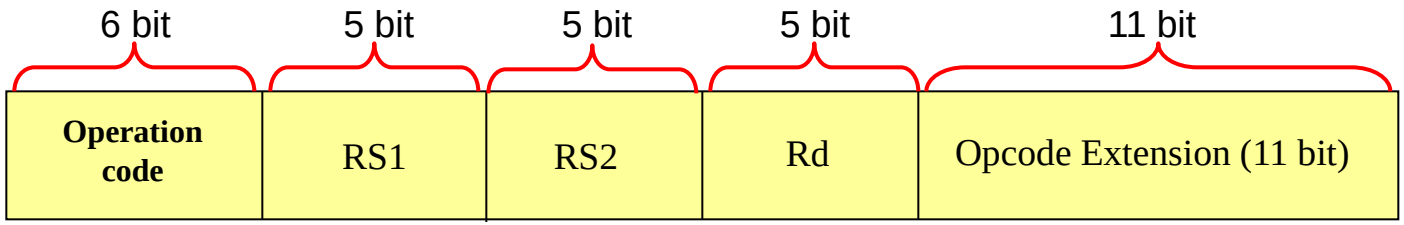
In the DLX architecture, the interrupt return address is saved in a specific register (**IAR**). There is no STACK, so it is not possible to handle nested interrupts.

As we will see, interrupts are **detected** before the FETCH stage of the next instruction. In case an interrupt request is asserted and interrupts are enabled (**IEN** flag), the control unit saves the current PC into IAR, then it loads "0" onto the PC (hence, 0 is the address of the interrupt handler). Interrupts are then automatically disabled.

The function of the **RFE** instruction is to enable interrupts and to restore the PC with the current value from IAR (so it must always be the last instruction of the interrupt handler routine).

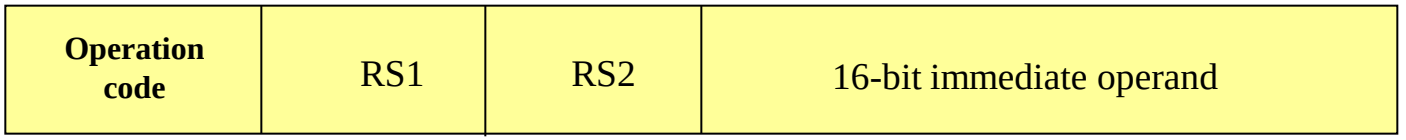
DLX Instruction Formats

R



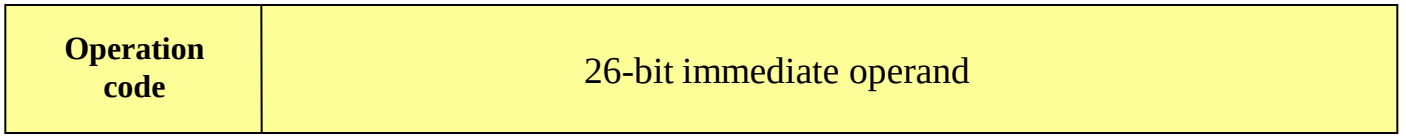
- **ALU instructions** in the form of: Rd ← Rs1 op Rs2

I



- **Load/Store** (RS1 = memory address; RS2 = destination register/source; *immediate operand: offset component of the memory address*)
- **Branch** (RS1 = source; RS2 not used; *immediate operand* = offset wrt. PC)
- **Jump Register** (JR) (RS2 : not used, RS1 = register copied on PC, *immediate operand*: not used)
- **Jump and Link register** (JALR) (RS2: not used, RS1 = register copied on PC, *immediate operand: not used*)
- **ALU with immediate operand** (RS2 ← RS1 op *immediate operand*)

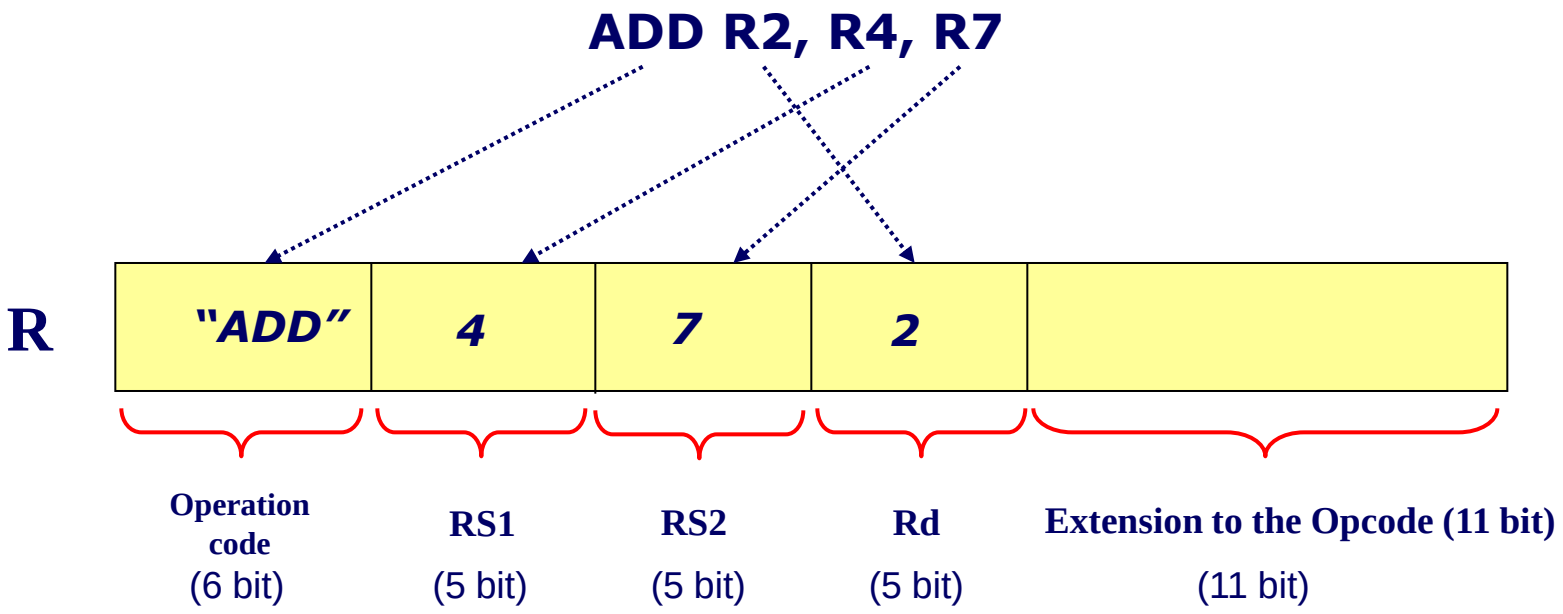
J



- **Jump, Jump and Link:** (J ,JAL) (*immediate operand*= offset wrt. PC)

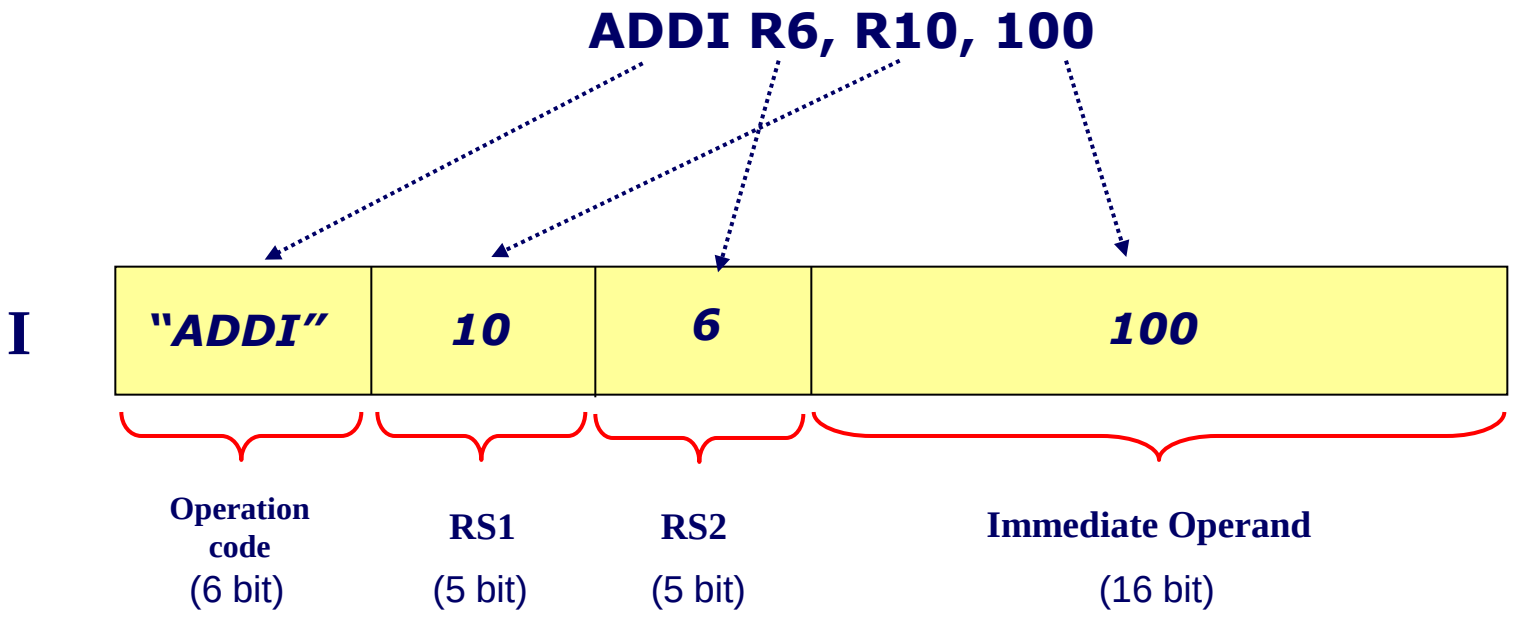
Encoding of R-format instructions

ALU instructions with 3 register operands



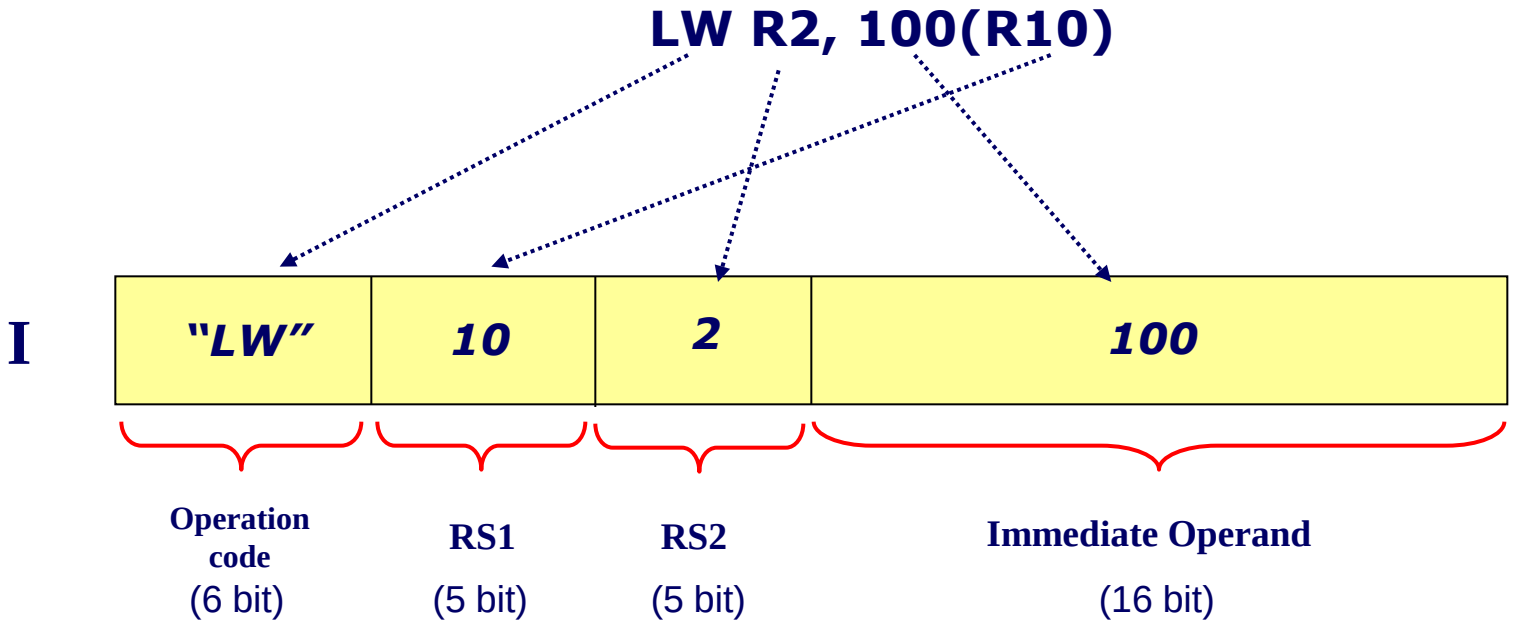
Rd ← Rs1 op Rs2

Encoding of I-format instructions (*ALU with immediate*)



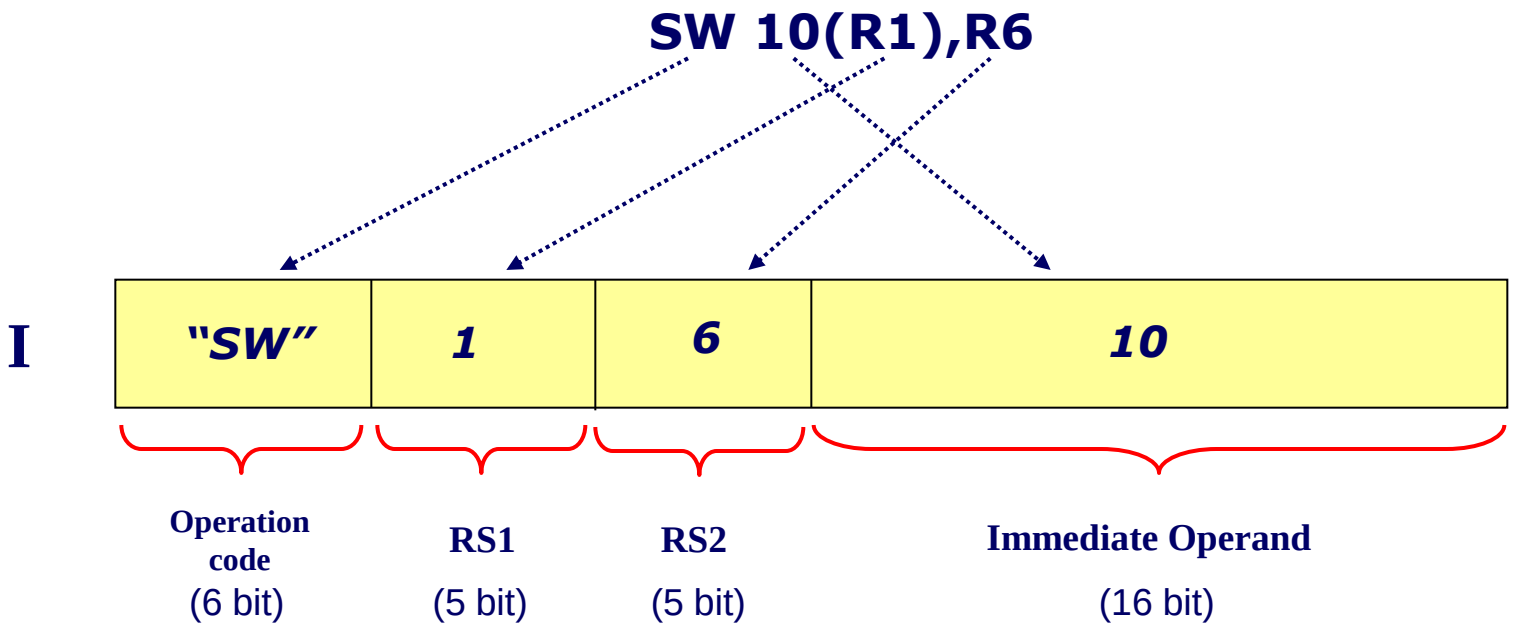
Rs2 ← Rs1 op immediate operand

Encoding of I-format instructions (*LOAD/STORE)*



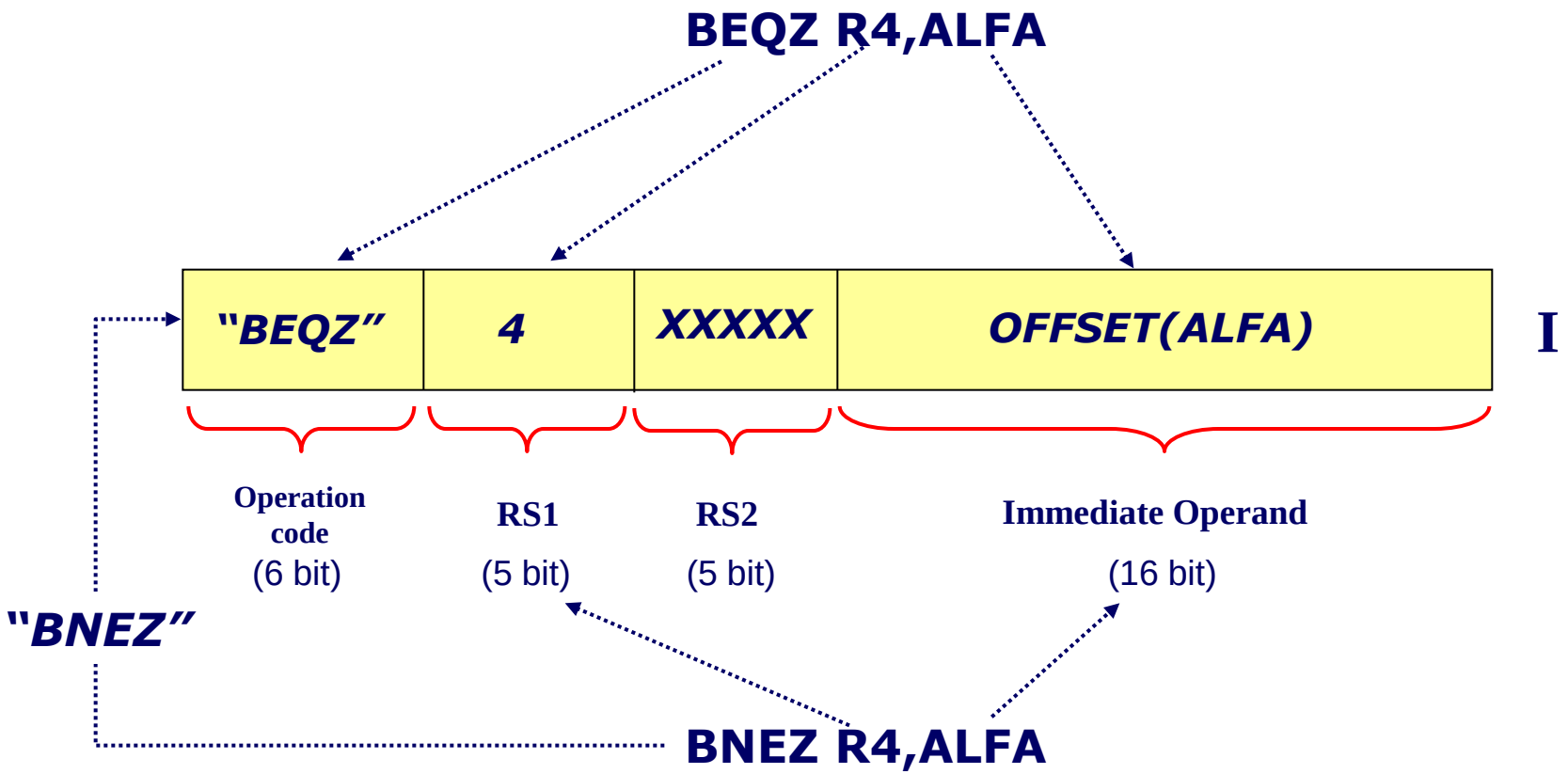
Rs2 ← M[Rs1 + immediate]

Encoding of I-format instructions (*LOAD/STORE)*



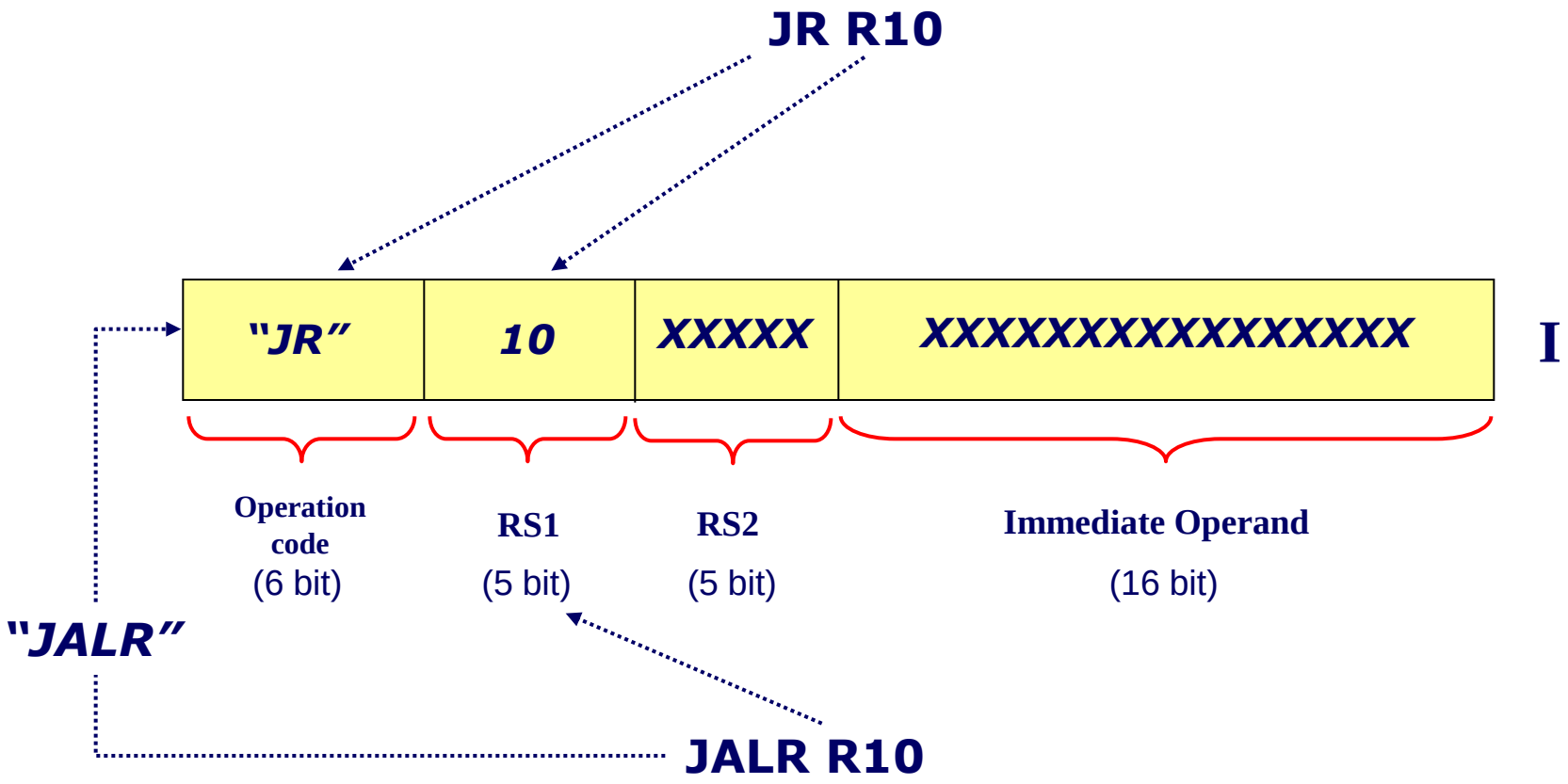
M[Rs1 + immediate] ← Rs2

Encoding of I-format instructions (*BRANCH*)



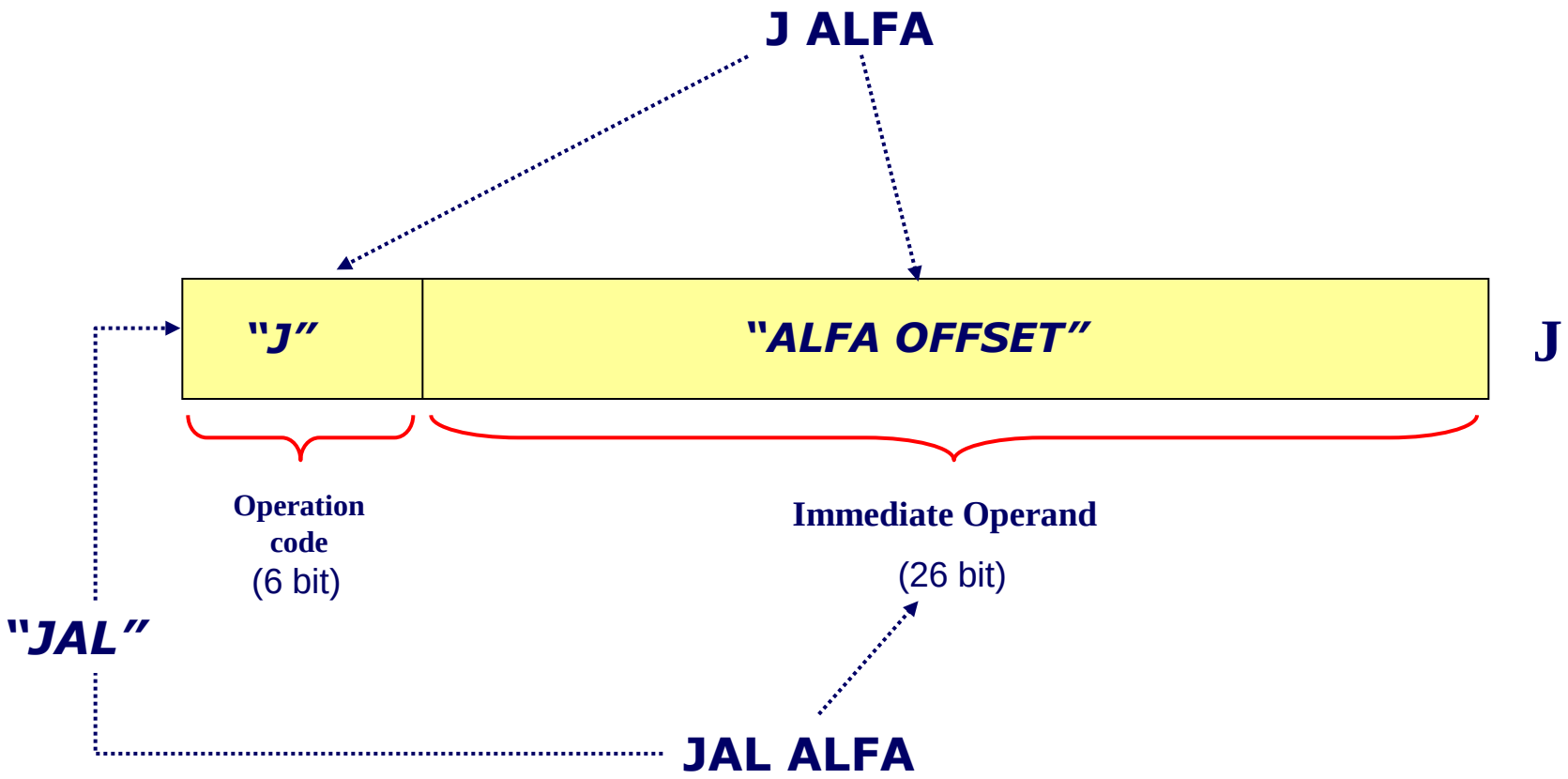
BEQZ:	**if (Rs1=0) then PC ← PC + immediate**
BNEZ:	**if (Rs1≠0) then PC ← PC + immediate**

Encoding of the I-format instructions (*JR, JALR*)



JR:	**PC ← Rs1**
JALR:	**R31 ← PC, PC ← Rs1**

Encoding of J-format instructions (*J, JAL*)



J:	**PC ← PC + OFFSET(ALFA)**
JAL:	**R31 ← PC, PC ←PC + OFFSET(ALFA)**

Example of DLX assembly code

- Write the DLX assembly code to compute the sum of the elements of a vector A of 8 elements. Use the following registers:
  - R1 for the current sum
  - R2 for the index
  - R3 for the iterations counter

```
        ADD     R1, R0, R0          ;reset the current sum
        ADD     R2, R0, R0          ;R2 equals index * 4  (4 is the number of bytes in a word)
        ADDI    R3, R0, 8           ;initialize the iterations counter

LOOP:   LW      R4, A(R2)           ; memory address of the operand computed at run-time
        ADD     R1, R1, R4          ; updates the current sum
        ADDI    R2, R2, 4           ; updates the index
        ADDI    R3, R3, -1          ; decrease the iterations counter
        BNEZ    R3, LOOP ; have all the 8 iterations been executed ?
        SW      Z(R0), R1           ; Z is the variable containing the result
        .....
```

Exercises

- With reference to the previous example of DLX code:
  - Identify the format of all instructions;
  - Draw the memory address space of the DLX and place code and data into it;
  - Draw a chart showing the behaviour of the PC during program execution;
  - List the values taken by R1 during program execution;
  - Compute the number of bus cycles carried out by the CPU during program execution, subdividing them between FETCH cycles and EXECUTE cycles.