B38DF: Computer Architecture and Embedded Systems

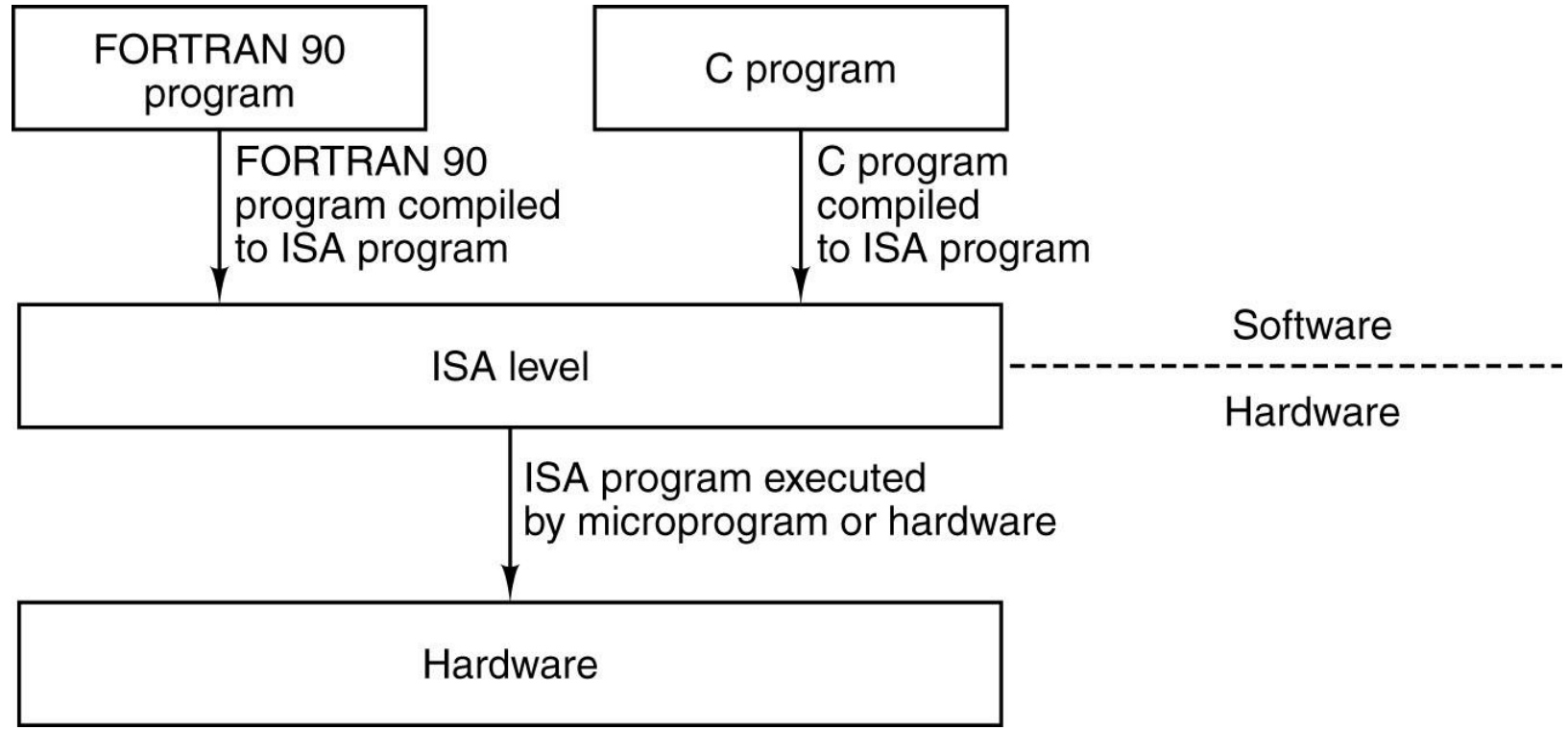
Alexander Belyaev

Heriot-Watt University
School of Engineering & Physical Sciences
Electrical, Electronic and Computer Engineering
Room: EM 2.29
E-mail: a.belyaev@hw.ac.uk

Based on the slides prepared by Dr. Mustafa Suphi Erden

The Instruction Set Architecture (ISA) Level

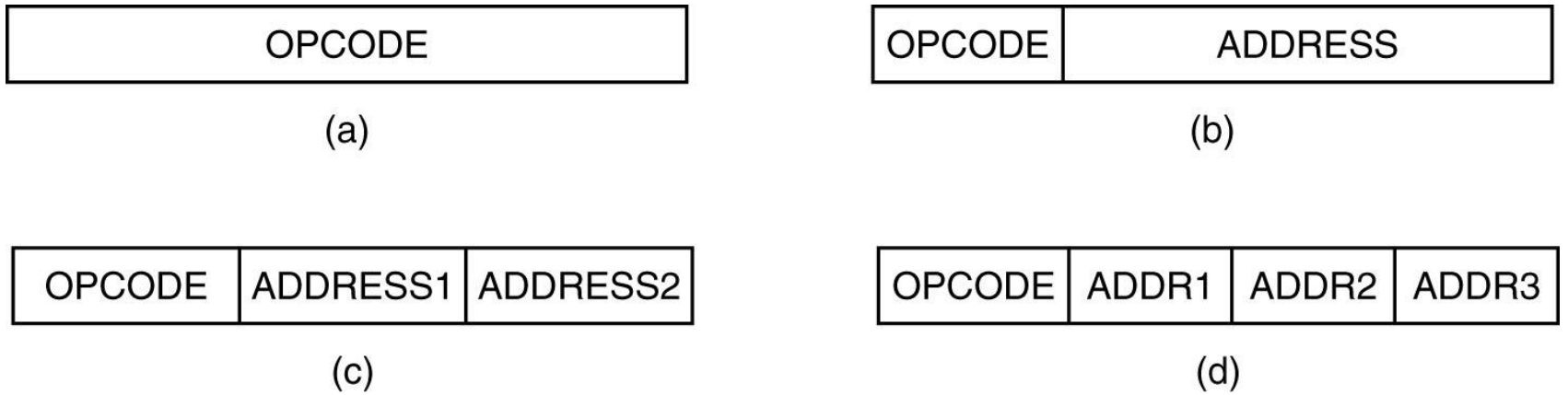
The ISA level is the interface between the compilers and the hardware.



ISA Level

- The ISA level defines the **interface between the compilers and the hardware**.
- ISA level is the **language both the compiler and hardware understands**.
- Programs in various high-level languages are translated to the common intermediate form – the ISA-level.
- ISA-level code is **what a compiler outputs** (ignoring operating system calls and symbolic assembly language).
- **Hardware is built to execute ISA-level programs** directly.

Instruction Formats (1)

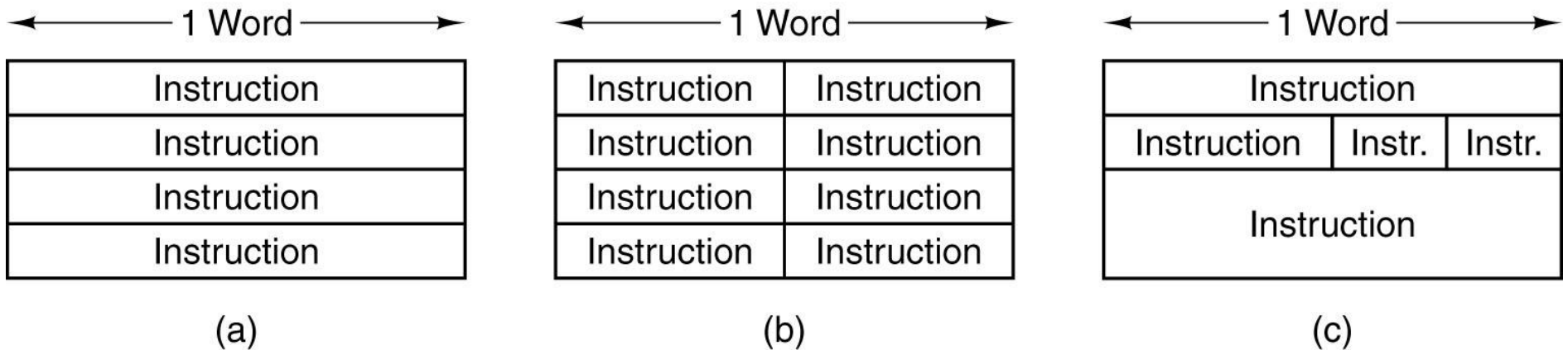


Four common instruction formats:

(a) Zero-address instruction. (b) One-address instruction

(c) Two-address instruction. (d) Three-address instruction.

Instruction Formats (2)



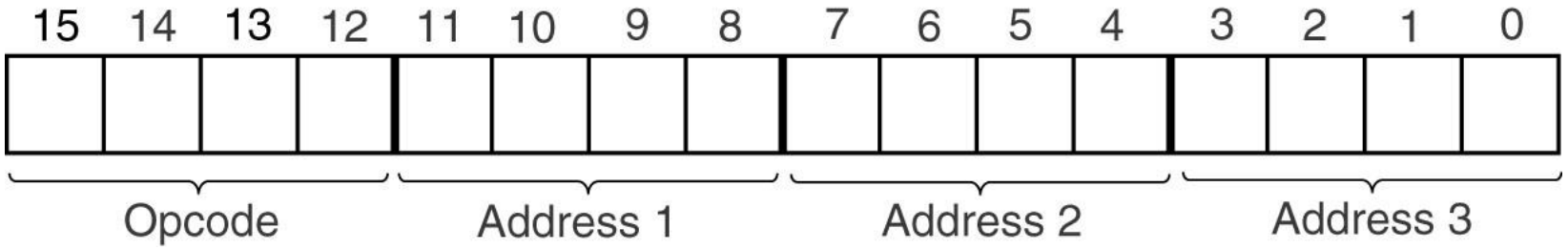
Some possible relationships between instruction and word length.

Trade-offs between Opcodes and Addresses

- (*n*+*k*) bit instruction: ***k*-bit opcode** + ***n*-bit address**
  - 2^k different instructions + 2^n addressable memory cells.
- (*n*+*k*) bit instruction: **(*k*–*1*)-bit opcode** + **(*n*+*1*)-bit address**
  - half as many instructions+ twice as much memory address
- A way of solving this trade-off is using **expanding opcode**.

Expanding Opcodes (1)

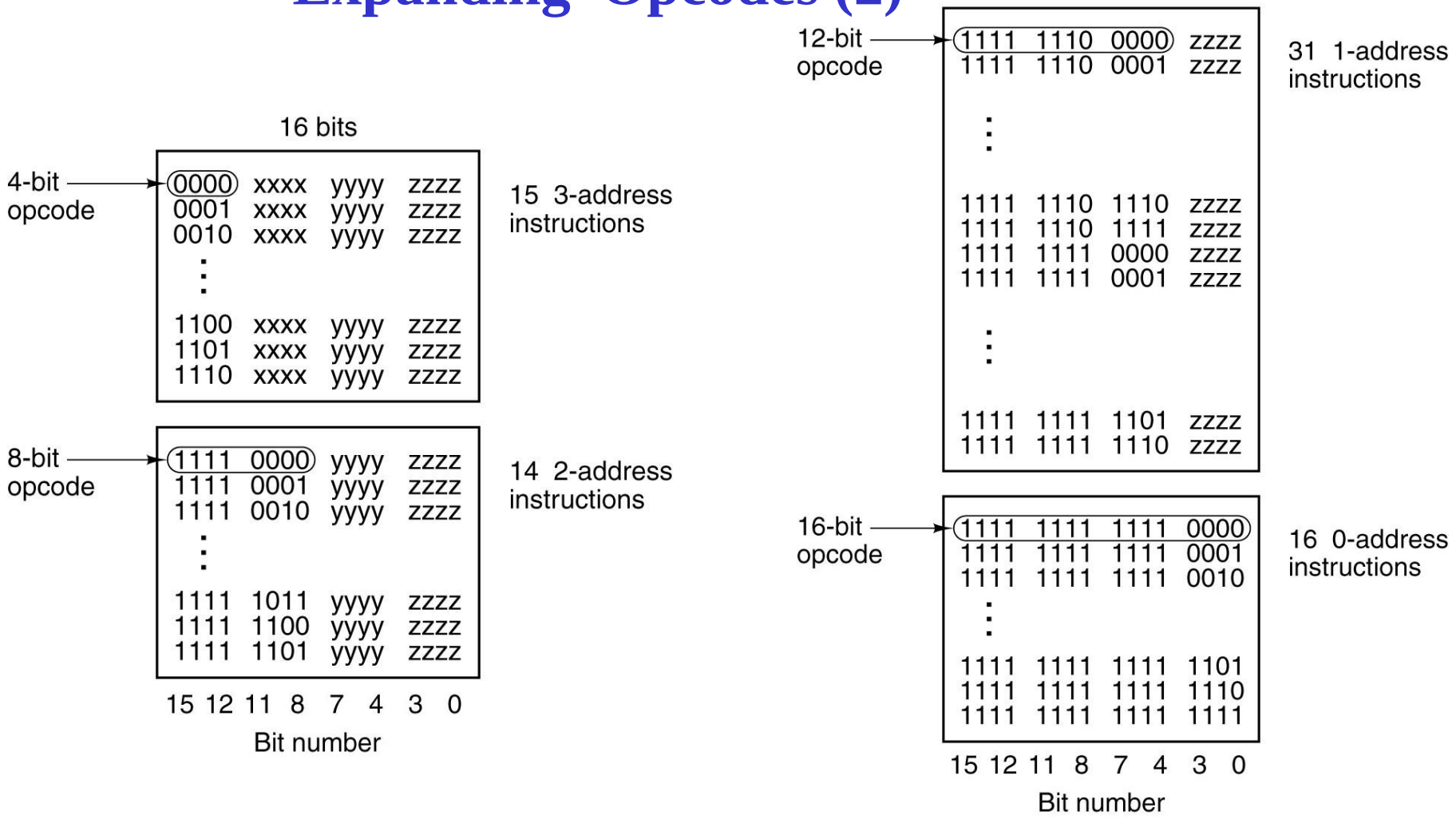
An instruction with a 4-bit opcode and three 4-bit address fields.



An instruction with a 4-bit opcode and three 4-bit address fields, giving

16 three-address instructions

Expanding Opcodes (2)



An expanding opcode allowing **15 three-address instructions**, **14 two-address instructions**, **31 one-address instructions**, and **16 zero-address instructions**. The fields marked *xxxx*, *yyyy*, and *zzzz* are 4-bit address fields.

Addressing

- A large portion of the bits in a program are used to specify **where operands come from** rather than what operations are being performed on them.

An Example for **E=(A+B)*(C+D)**

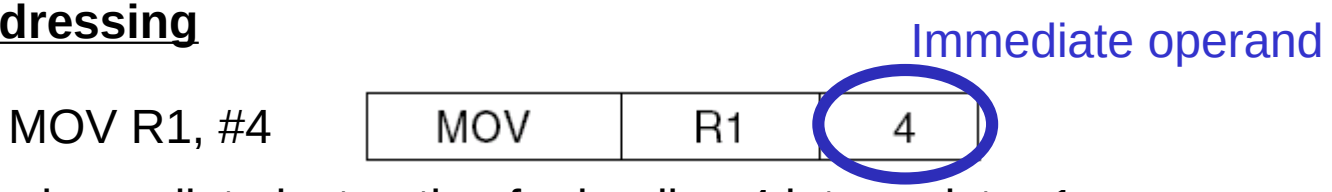
(R: registry; A-E: memory addresses)

```
LOAD    R1, A
ADD     R1, B
LOAD    R2, C
ADD     R2, D
MUL     R2, R1
STORE   E, R2
```

6 opcodes and 12 addressing are used in this program.

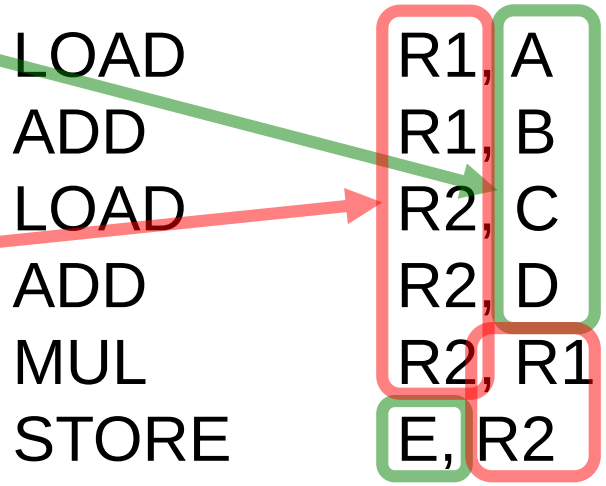
Basic Addressing Modes (1)

- **Immediate Addressing**



An immediate instruction for loading 4 into register 1.

- **Memory Direct Addressing**
  - Specifying an operand in the memory by giving its full address
- **Register Direct Addressing**
  - Specifying an operand in the register by giving its full address



Basic Addressing Modes (2)

- **Register Indirect Addressing**
  - The operand being specified comes from memory or goes to memory, but **its address is contained in a register**.
  - The register gives the address of the operand in the memory.
  - When an address is used in this manner, it is called a **pointer**.

```
        MOV R1,#0          ; accumulate the sum in R1, initially 0
        MOV R2,#A          ; R2 = address of the array A
        MOV R3,#A+4096     ; R3 = address of the first word beyond A
LOOP:   ADD R1,(R2)        ; register indirect through R2 to get operand
        ADD R2,#4          ; increment R2 by one word (4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

A generic assembly program for computing the sum of the elements of an array.

Basic Addressing Modes (3)

- **Indexed Addressing**
  - Addressing memory by giving a register (explicit or implicit) plus a constant offset.

MOV R4, A(R2)

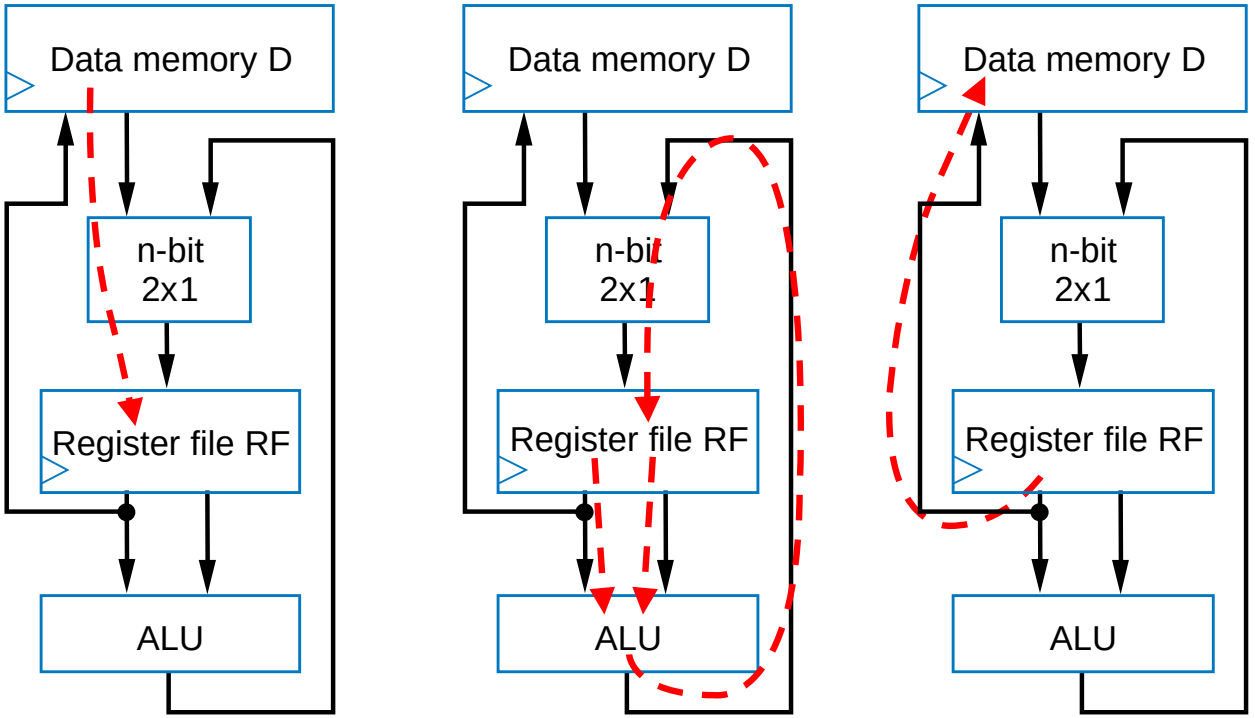
R4 = **A [i]**

```
        MOV R1,#0          ; accumulate the OR in R1, initially 0
        MOV R2,#0          ; R2 = index, i, of current product: A[i] AND B[i]
        MOV R3,#4096       ; R3 = first index value not to use
LOOP:   MOV R4,A(R2)       ; R4 = A[i]
        AND R4,B(R2)       ; R4 = A[i] AND B[i]
        OR R1,R4           ; OR all the Boolean products into R1
        ADD R2,#4          ; i = i + 4 (step in units of 1 word = 4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

A generic assembly program for computing the OR of Ai AND Bi for two 1024-element arrays.

Basic Datapath Operations

- **Load** operation: Load data from data memory to RF
- **ALU** operation: Transforms data by passing one or two RF register values through ALU, performing operation (ADD, SUB, AND, OR, etc.), and writing back into RF.
- **Store** operation: Stores RF register value back into data memory
- Each operation can be done in one clock cycle



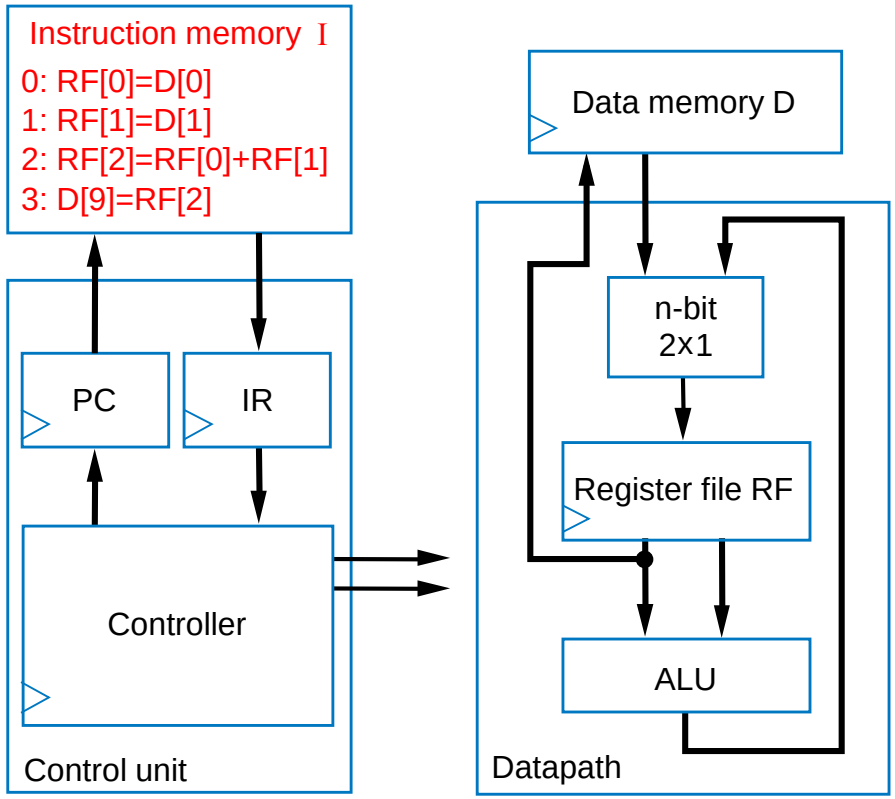
Load operation ALU operation Store operation

Basic Architecture – Control Unit

- D[9] = D[0] + D[1] – requires a sequence of four datapath operations:

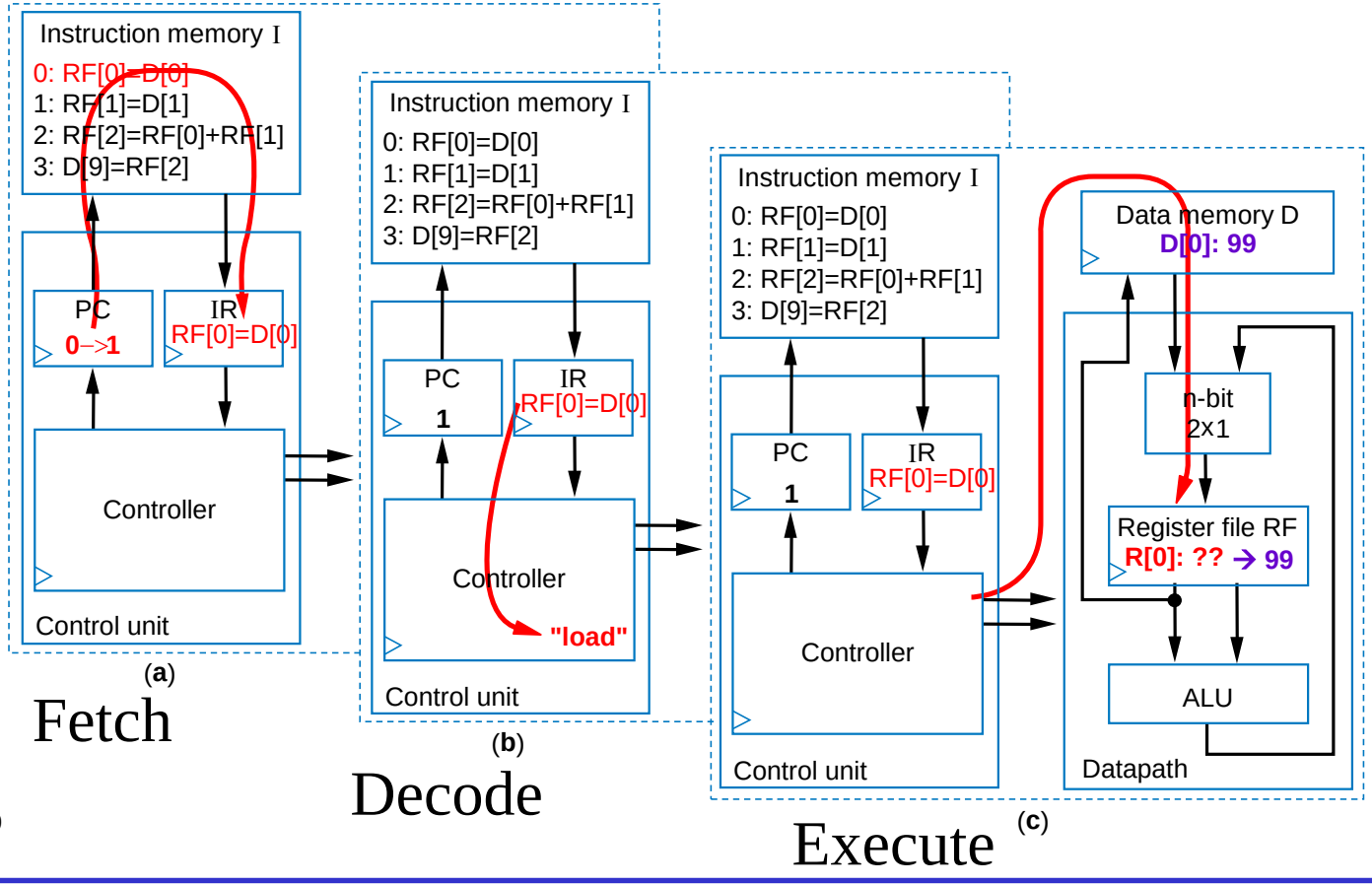
  0: RF[0] = D[0]
  1: RF[1] = D[1]
  2: RF[2] = RF[0] + RF[1]
  3: D[9] = RF[2]

- Each operation is an *instruction*
  - Sequence of instructions – *program*
  - Looks cumbersome, but that's the world of programmable processors – Decomposing desired computations into processor-supported operations
  - Store program in *Instruction memory*
  - *Control unit* reads each instruction and executes it on the datapath
    - PC: Program counter – address of current instruction
    - IR: Instruction register – current instruction

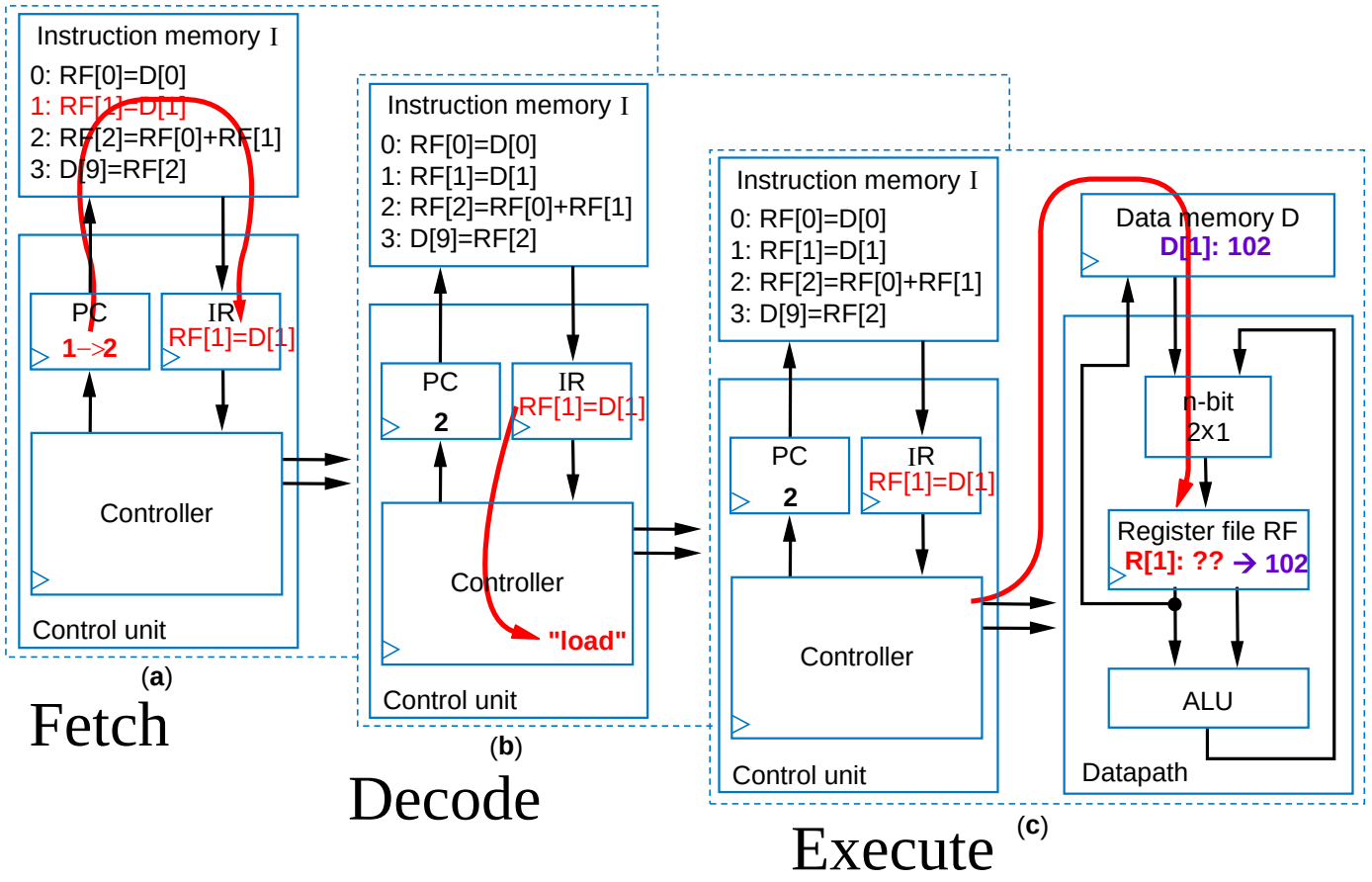


Basic Architecture – Control Unit

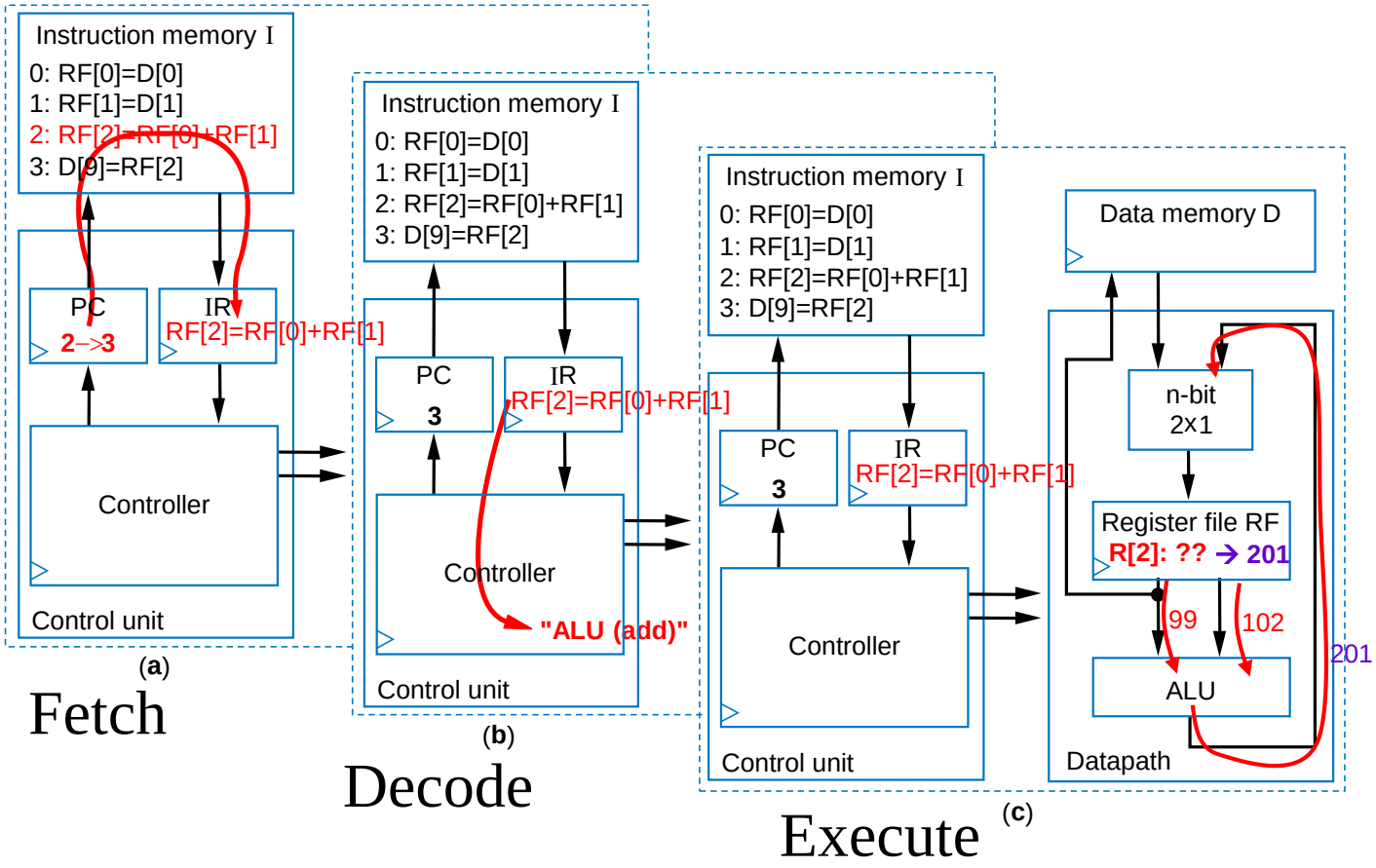
- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath



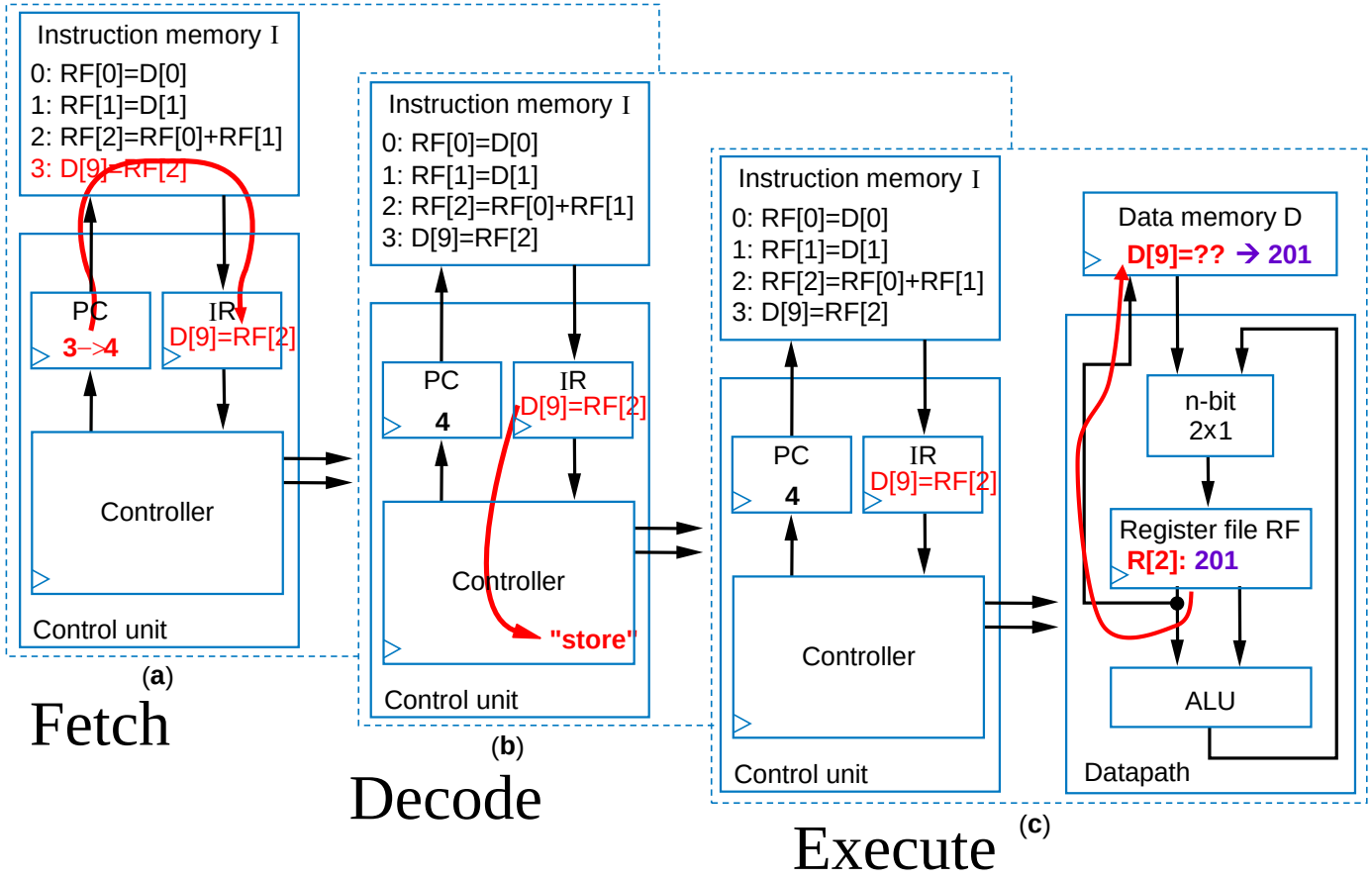
Basic Architecture – Control Unit



Basic Architecture – Control Unit



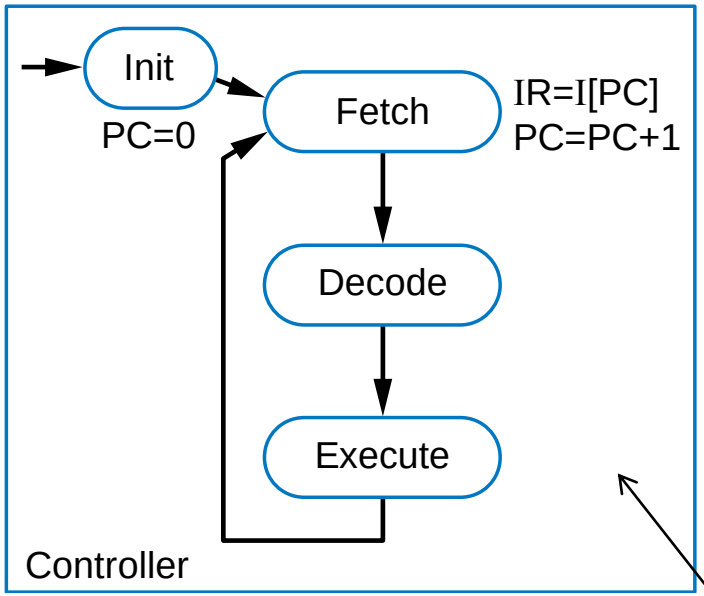
Basic Architecture – Control Unit

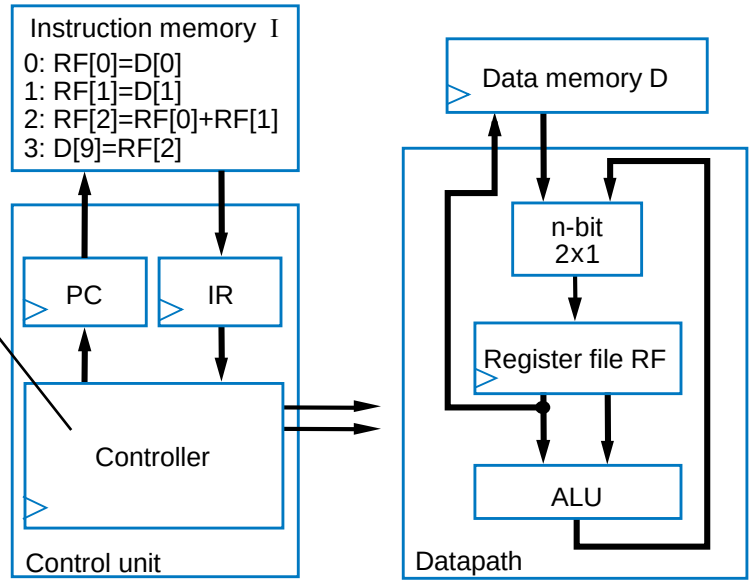


Basic Architecture – Control Unit

To summarize, the control unit processes each instruction in three stages:

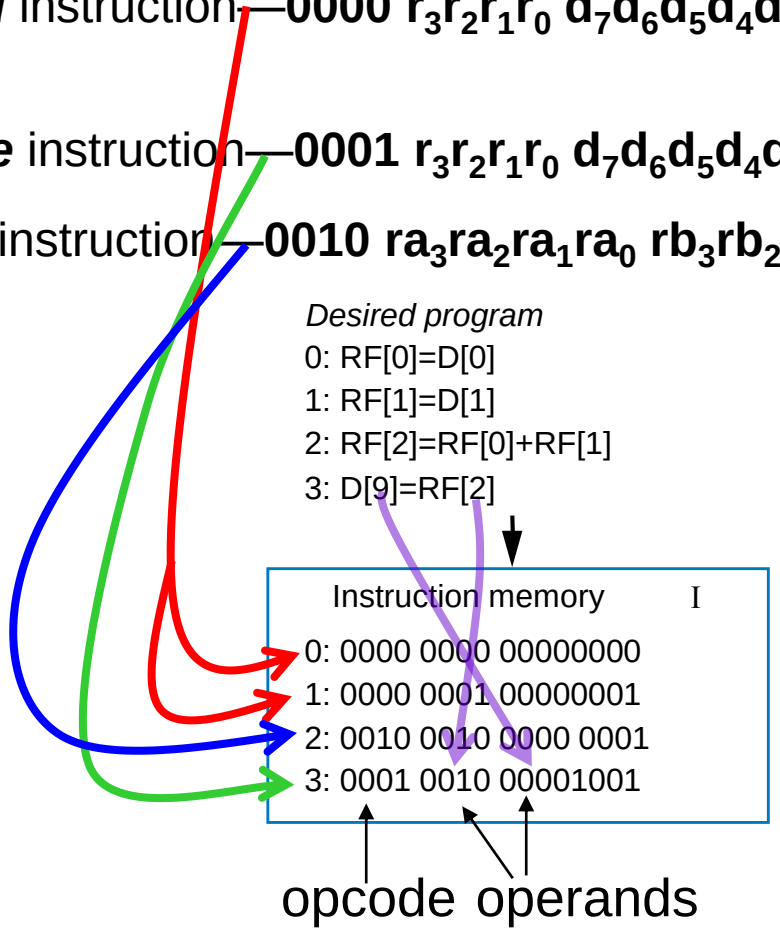
1. first *fetching* the instruction by loading the current instruction into *IR* and incrementing the *PC* for the next fetch,
2. next *decoding* the instruction to determine its operation, and
3. finally *executing* the operation by setting the appropriate control lines for the datapath, if applicable. If the operation is a datapath operation, the operation may be one of three possible types:
   (a) *loading* a data memory location into a register file location,
   (b) transforming data using an *ALU* operation on register file locations and writing results back to a register file location, or
   (c) *storing* a register file location into a data memory location.





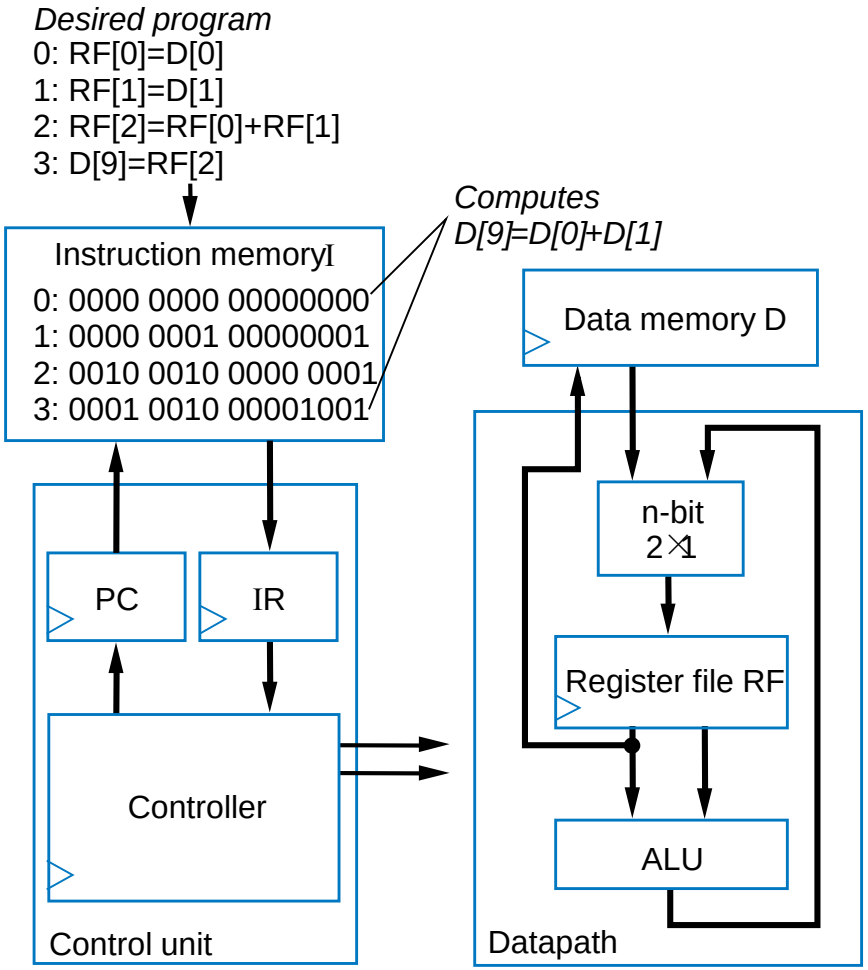
Three-Instruction Programmable Processor

- Instruction Set – List of allowable instructions and their representation in memory, e.g.,
  - ***Load*** instruction—**0000 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**
  - ***Store*** instruction—**0001 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**
  - ***Add*** instruction—**0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**



Instructions in 0s and 1s – *machine code*

Program for Three-Instruction Processor



Program for Three-Instruction Processor

- Another example program in machine code
  - Compute D[5] = D[5] + D[6] + D[7]

```
0: 0000 0000 00000101  // RF[0] = D[5]
1: 0000 0001 00000110  // RF[1] = D[6]
2: 0000 0010 00000111  // RF[2] = D[7]
3: 0010 0000 0000 0001  // RF[0] = RF[0] + RF[1]
                        // which is D[5]+D[6]
4: 0010 0000 0000 0010  // RF[0] = RF[0] + RF[2]
                        // now D[5]+D[6]+D[7]
5: 0001 0000 00000101  // D[5] = RF[0]
```

–***Load*** instruction—**0000 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**

–***Store*** instruction—**0001 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**

–***Add*** instruction—**0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

Assembly Code

- Machine code (0s and 1s) hard to work with
- Assembly code – Uses mnemonics
  - ***Load*** instruction**—MOV Ra, d**
    - specifies the operation *RF[a]=D[d]*. *a* must be 0,1, ..., or 15—so *R0* means *RF[0]*, *R1* means *RF[1]*, etc. *d* must be 0, 1, ..., 255
  - ***Store*** instruction**—MOV d, Ra**
    - specifies the operation *D[d]=RF[a]*
  - ***Add*** instruction**—ADD Ra, Rb, Rc**
    - specifies the operation *RF[a]=RF[b]+RF[c]*

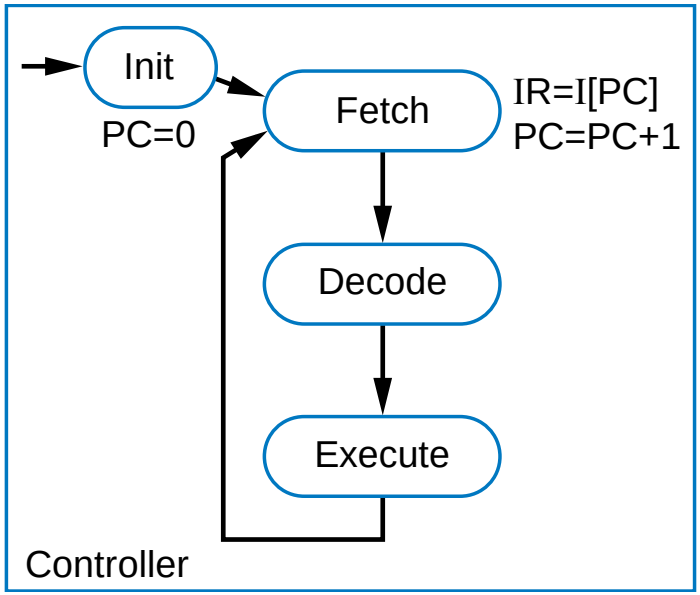
Desired program

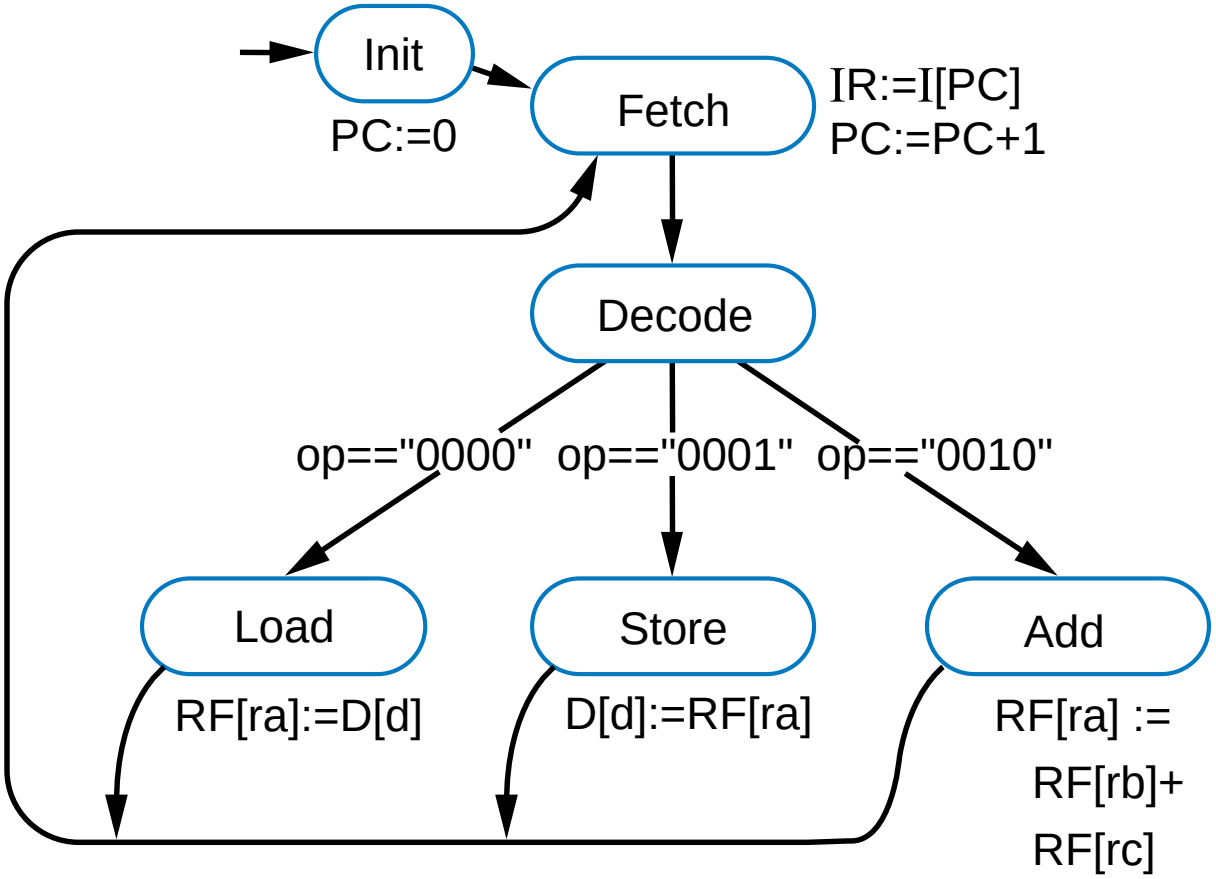
0: RF[0]=D[0]
1: RF[1]=D[1]
2: RF[2]=RF[0]+RF[1]
3: D[9]=RF[2]

machine code	assembly code
0: 0000 0000 00000000	0: MOV R0, 0
1: 0000 0001 00000001	1: MOV R1, 1
2: 0010 0010 0000 0001	2: ADD R2, R0, R1
3: 0001 0010 00001009	3: MOV 9, R2

Control-Unit and Datapath for Three-Instruction Processor

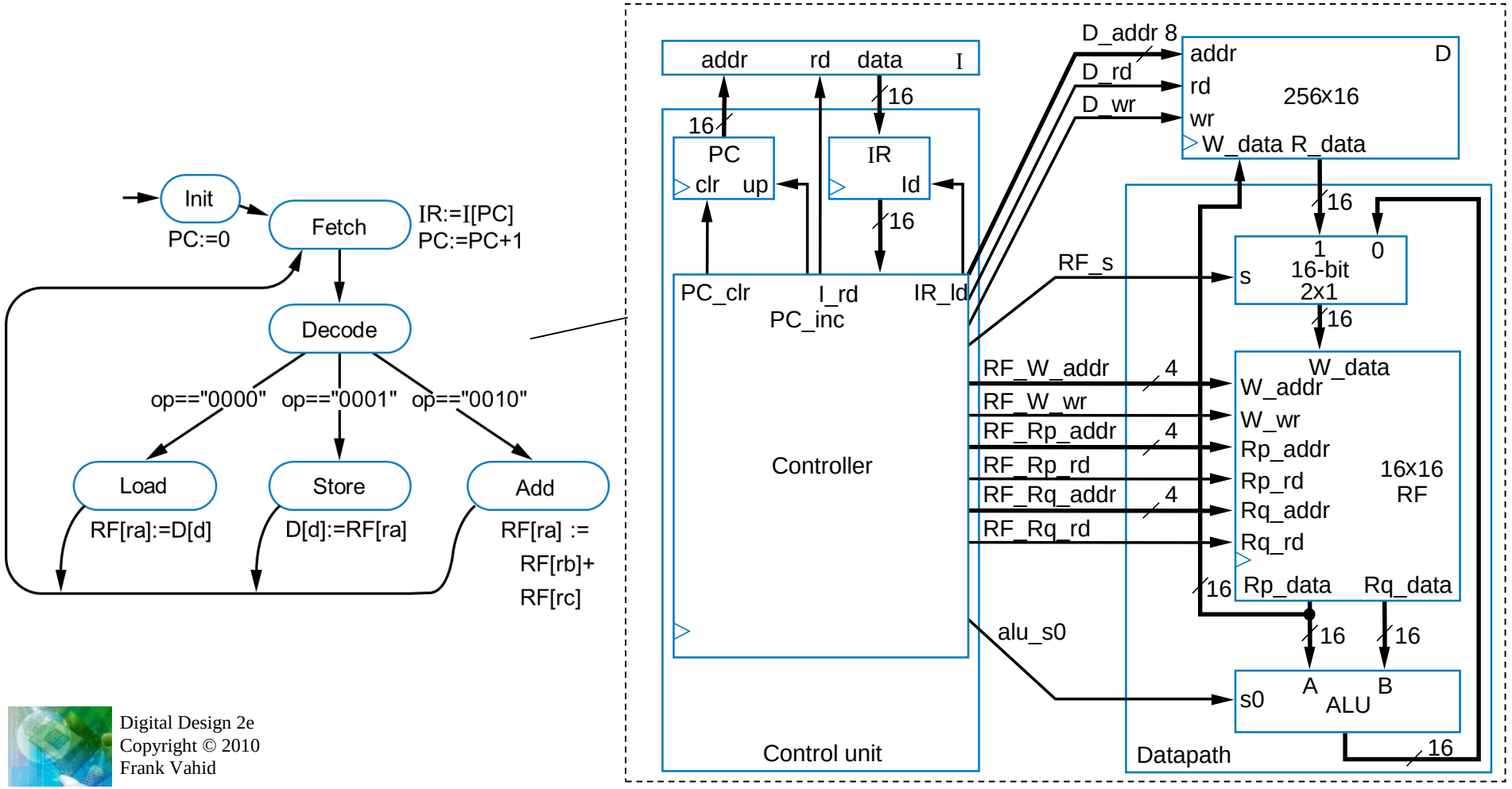
- To design the processor, we can begin with a high-level state machine description of the processor's behavior





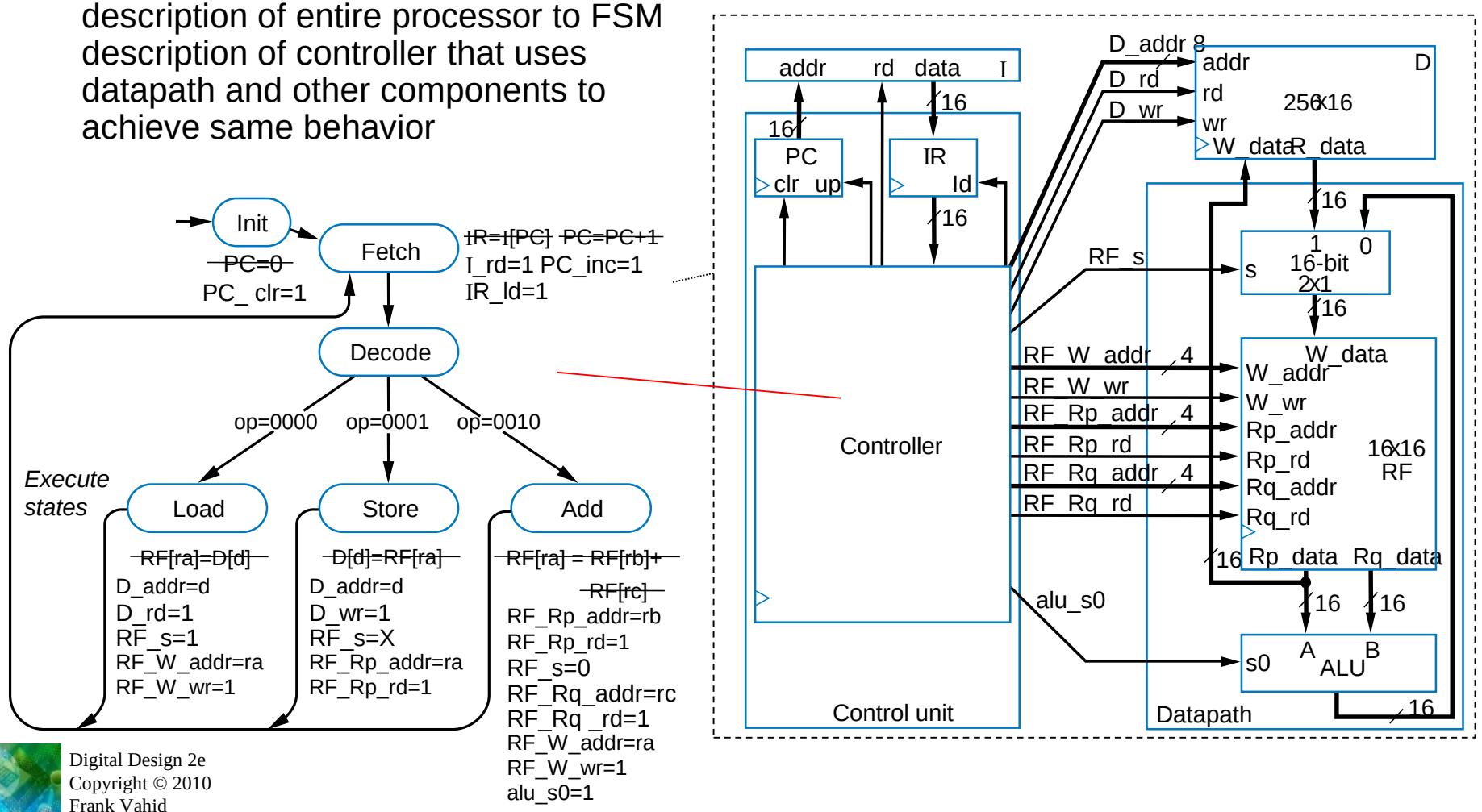
Control-Unit and Datapath for Three-Instruction Processor

- Create detailed connections among components



Control-Unit and Datapath for Three-Instruction Processor

- Convert high-level state machine description of entire processor to FSM description of controller that uses datapath and other components to achieve same behavior



A Six-Instruction Programmable Processor

Let us add three more instructions:

–***Load-constant*** instruction—**0011 $r_3r_2r_1r_0$ $c_7c_6c_5c_4c_3c_2c_1c_0$**

MOV Ra, #c—specifies the operation *RF[a]=c*

–***Subtract*** instruction—**0100 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

SUB Ra, Rb, Rc—specifies the operation *RF[a]=RF[b] – RF[c]*

–***Jump-if-zero*** instruction—**0101 $ra_3ra_2ra_1ra_0$ $o_7o_6o_5o_4o_3o_2o_1o_0$**

JMPZ Ra, offset—specifies the operation *PC = PC + offset* if *RF[a]* is 0

TABLE 8.1 Six-instruction instruction set..

Instruction	Meaning
MOV Ra, d	RF[a] = D[d]
MOV d, Ra	D[d] = RF[a]
ADD Ra, Rb, Rc	RF[a] = RF[b]+RF[c]
MOV Ra, #C	RF[a] = C
SUB Ra, Rb, Rc	RF[a] = RF[b]-RF[c]
JMPZ Ra, offset	PC=PC+offset if RF[a]=0

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

A Six-Instruction Programmable Processor

Example program – Count number of non-zero words in D[4] and D[5]
- Result will be either 0, 1, or 2
- Put result in D[9]

(a)	(b)
MOV R0, #0; // initialize result to 0	0011 0000 00000000
MOV R1, #1; // constant 1 for incrementing result	0011 0001 00000001
MOV R2, 4; // get data memory location 4	0000 0010 00000100
JMPZ R2, lab1; // if zero, skip next instruction	0101 0010 00000010
ADD R0, R0, R1; // not zero, so increment result	0010 0000 0000 0001
lab1:MOV R2, 5; // get data memory location 5	0000 0010 00000101
JMPZ R2, lab2; // if zero, skip next instruction	0101 0010 00000010
ADD R0, R0, R1; //not zero, so increment result	0010 0000 0000 0001
lab2:MOV 9, R0; // store result in data memory location 9	0001 0000 00001001

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

A Six-Instruction Programmable Processor: A loop example

```
i=0;
sum=0;
while ( i!=N ) {
   sum = sum + i;
   i = i + 1;
}
```

N is stored at D[9]

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

A Six-Instruction Programmable Processor: A loop example

```
i=0;
sum=0;
while ( i!=N ) {
   sum = sum + i;
   i = i + 1;
}
```

N is stored at D[9]

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

```
        MOV R0, #0        // R0 is "i"
        MOV R1, #0        // R1 is "sum"
        MOV R2, #1        // R2 is the constant "1"
        MOV R3, 9         // R3 is "N" or "D[9]"
        MOV R4, #0        // R4 is the constant "0" (for looping)
loop:   SUB R5, R3, R0    // R5 = N - i
        JMPZ R5, done     // if i==N, end while loop
        ADD R1, R1, R0    // sum = sum + i
        ADD R0, R0, R2    // i = i + 1
        JMPZ R4, loop     // continue through while loop
done:
```