Computer Organization and Architecture

Prof. Indranil Sengupta

Dr. Sarani Bhattacharya

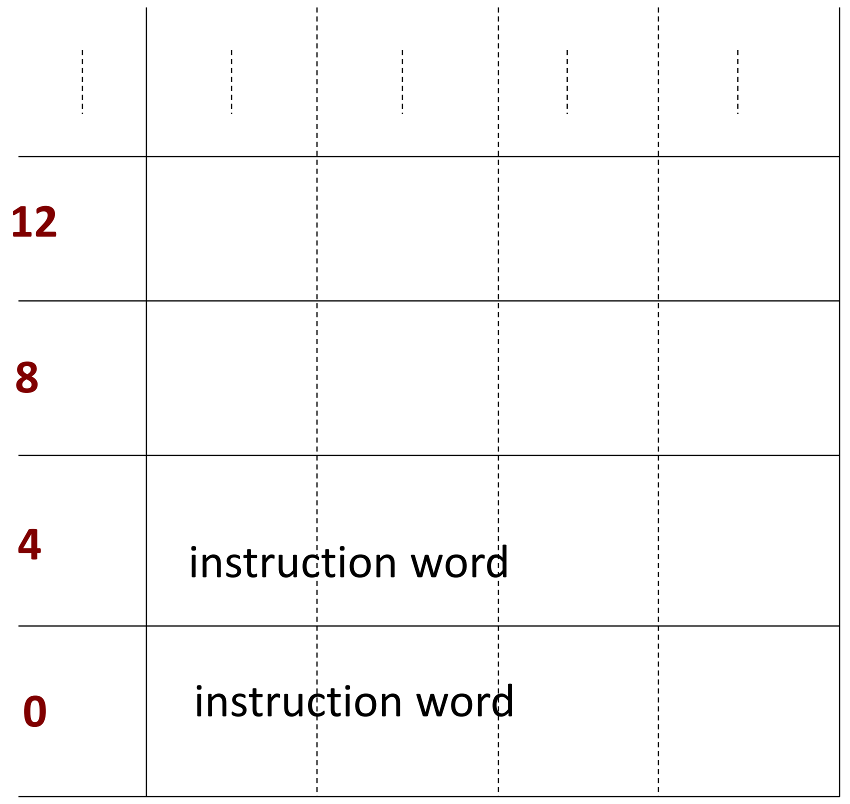
Department of Computer Science and Engineering

IIT Kharagpur

Design of Control Unit

Instructions

- Instructions are stored in main memory.
- Program Counter (PC) points to the next instruction.
    - MIPS instructions are 4 bytes (32 bits) long.
    - All instructions starts from an address that is multiple of 4 (last 2 bits 00).
    - Normally, PC is incremented by 4 to point to the next instruction.

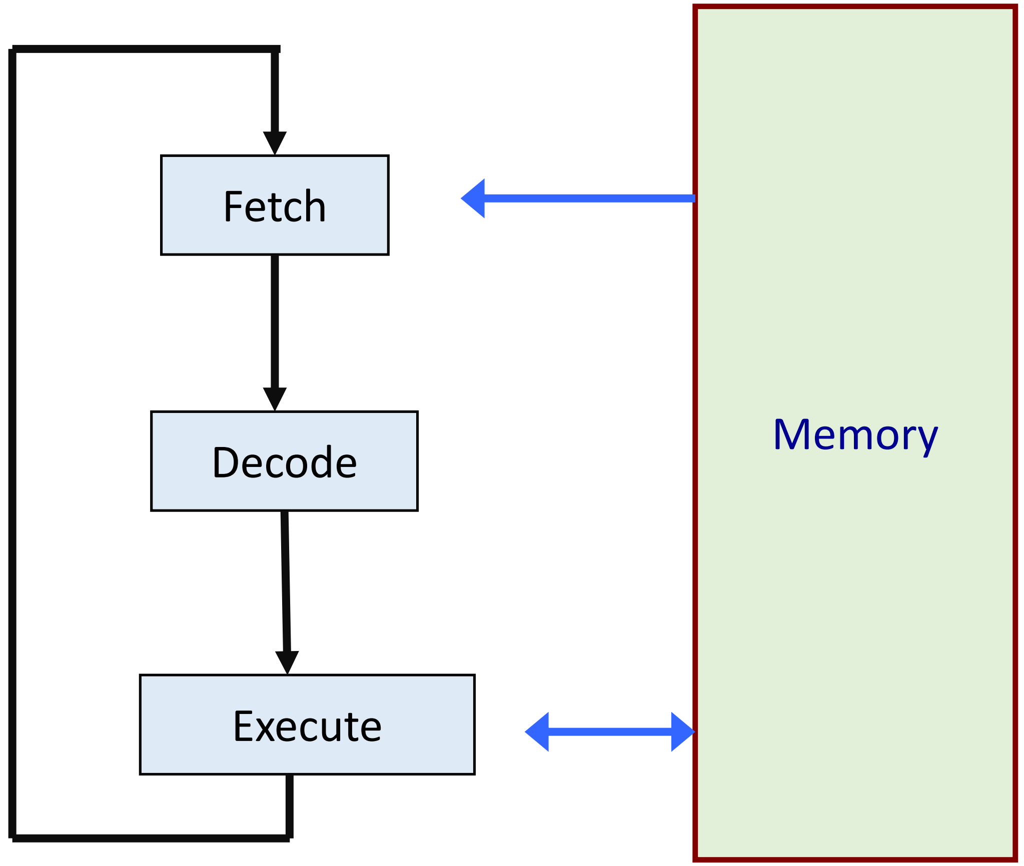


Addressing a Byte in Memory

- Each byte in memory has a unique address.
  - Memory is said to be *byte addressable*.
- Typically the instructions are of 4 bytes, hence the instruction memory is addressed in terms of 4 bytes (word length = 32 bits).
- When an instruction is executed, PC is incremented by 4 to point to the next instruction.

How an instruction Gets Executed?

```
repeat forever
      // till power off or
      // system failure
{
    Fetch instruction
    Decode instruction
    Execute instruction
}
```



The Fetch-Execute Cycle

- Fetch the next instruction from memory.
- Decode the instruction.
- Execution Cycle:
  - Gets data from memory if needed (data not available in the processor)
  - Perform the required operation on the data.
  - May also store the result back in memory or register.

Registers: IR and PC

- **Program Counter (PC)** holds the address of the memory location containing the next instruction to be executed.
- **Instruction Register (IR)** contains the current instruction being executed.
- Basic processing cycle to be implemented:
    - Instruction Fetch (IF)

      IR $\leftarrow$ Mem [PC]
    - Considering the word length of the machine is 32 bit, the PC is incremented by 4 to point to the next instruction.

      PC $\leftarrow$ PC + 4
    - Carry out the operations specified in IR.

Example: Add R1, R2

Address	Instruction
1000	ADD R1, R2
1004	MUL R3, R4

a)	PC	= 1000
b)	MAR	= 1000
c)	PC	= PC + 4 = 1004
d)	MDR	= “ADD R1, R2”
e)	IR	= “ADD R1, R2”
		(Decode and finally execute)
f)	R1	= R1 + R2

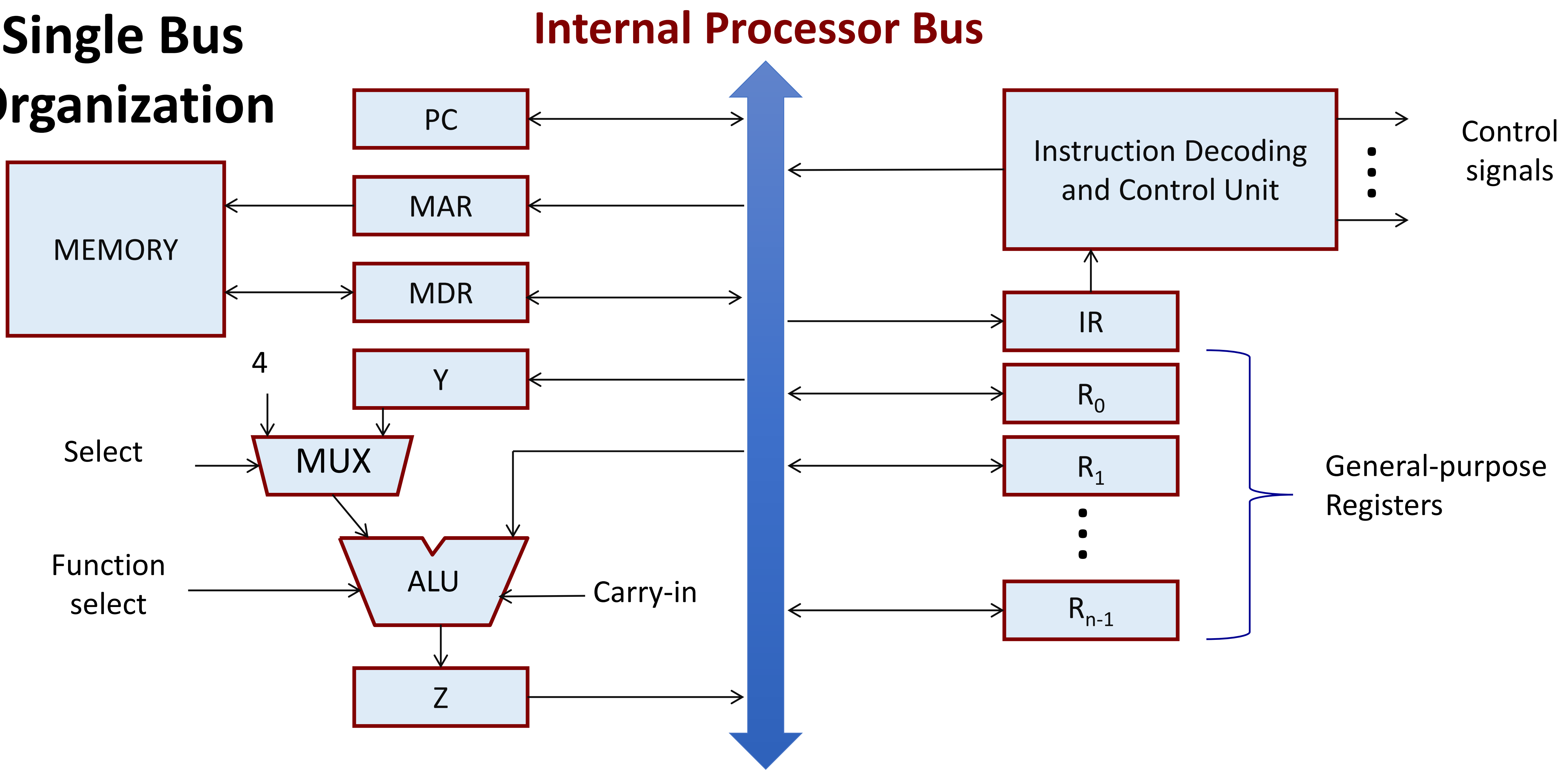
Requirement for Instruction Execution

- The necessary registers must be present.
- The internal organization of the registers must be known.
- The data path must be known.
- For instruction execution, a number of micro-operations are carried out on the data path.
  - An instruction consists of several micro-operations or micro-instructions.
  - May involve movement of data.

Kinds of Data Movement

- Broadly three types:
  a) Register to Register
  b) Register to ALU
  c) ALU to Register
- Data movement is supported in the data path by:
  - The Registers
  - The Bus (single or multiple)
  - The ALU temporary Register (Z)

Single Bus Organization



Single Internal Bus Organization

- All the registers and various units are connected using a single internal bus.
- Registers R_0-R_{n-1} are general-purpose registers used for various purposes.
- Y and Z are used for storing intermediate results and never used by instructions explicitly.
- The multiplexer selects either a constant 4 or output of register Y.
    - When PC is incremented, a constant 4 has to be added.

- The instruction decoder and control unit is responsible for performing the actions specified by the instruction loaded into IR.
- The decoder generates all the control signals in the proper sequence required to execute the instruction specified by the IR.
- The registers, the ALU and the interconnecting bus are collectively referred to as the *datapath*.

Kinds of Operations

- Transfer of data from one register to another.

```
MOVE   R1, R2
```

- Perform arithmetic or logic operation on data loaded into registers.

```
ADD    R1, R2
```

- Fetch the content of a memory location and load it into a register.

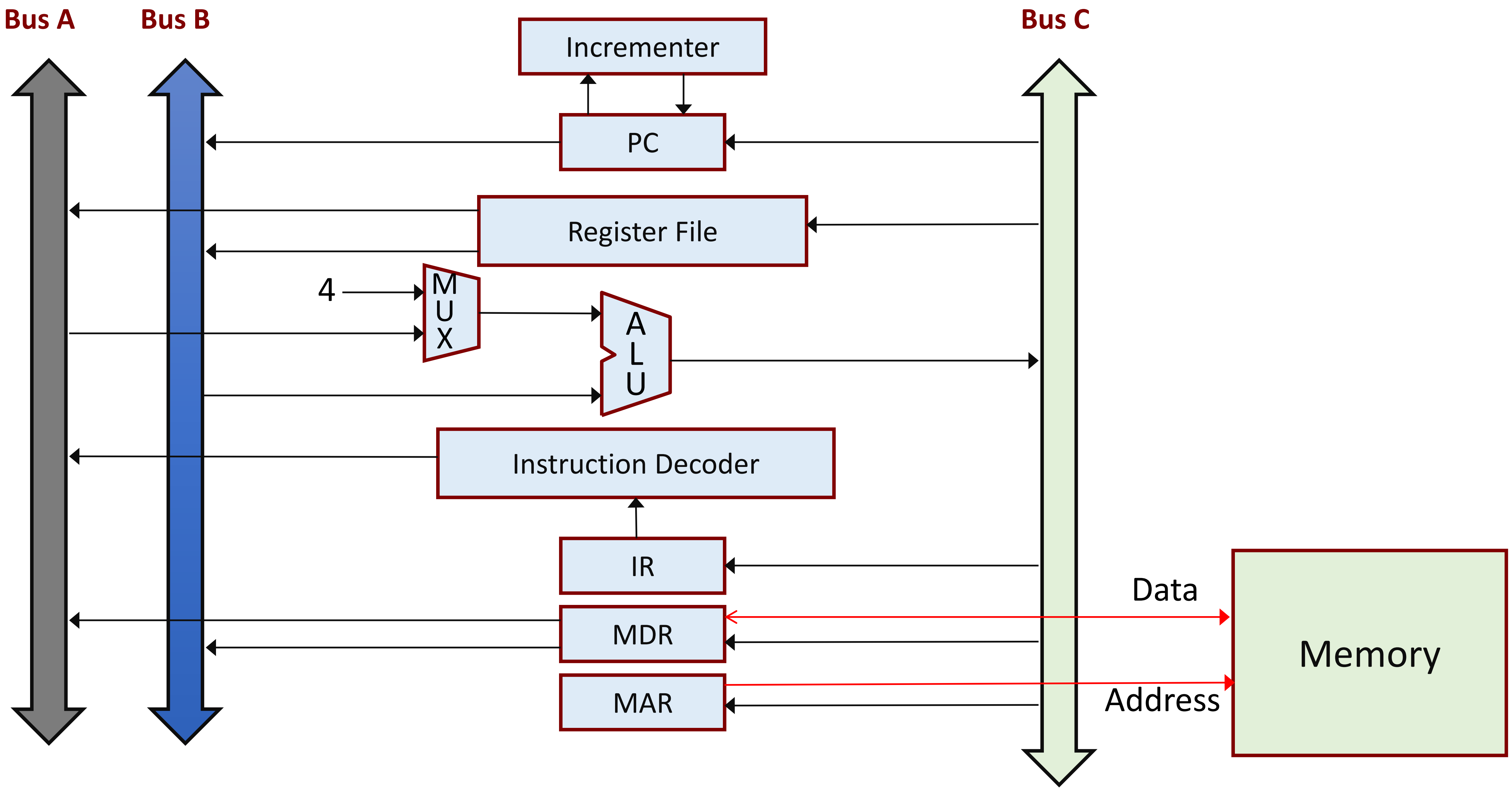
```
MOVE   R1, LOCA
```

- Store a word of data from a register into a given memory location.

```
MOVE   LOCA, R1
```

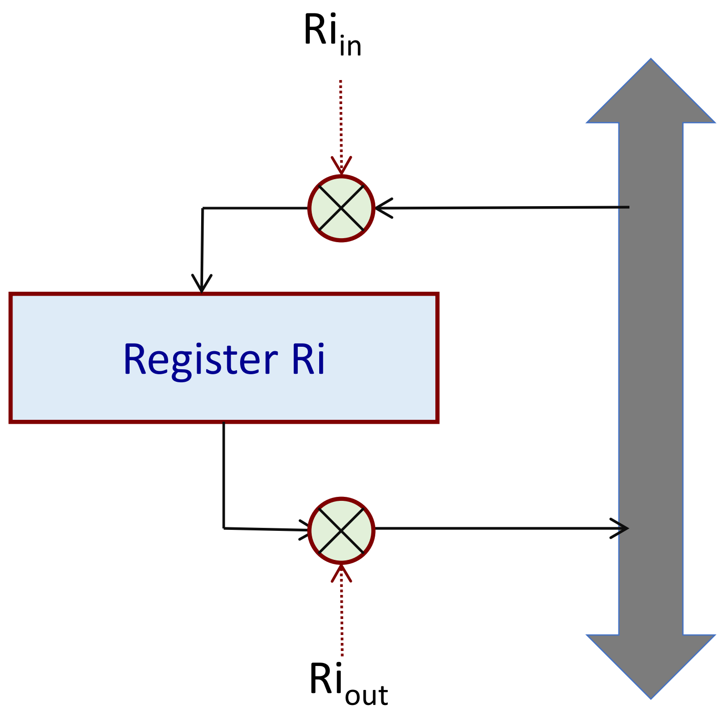
Three Bus Organization

- A typical 3-bus architecture for the processor datapath is shown in the next slide.
    - The 3-bus organization is internal to the CPU.
    - Three buses allow three parallel data transfer operations to be carried out.
- Less number of cycles required to execute an instruction compared to single bus organization.

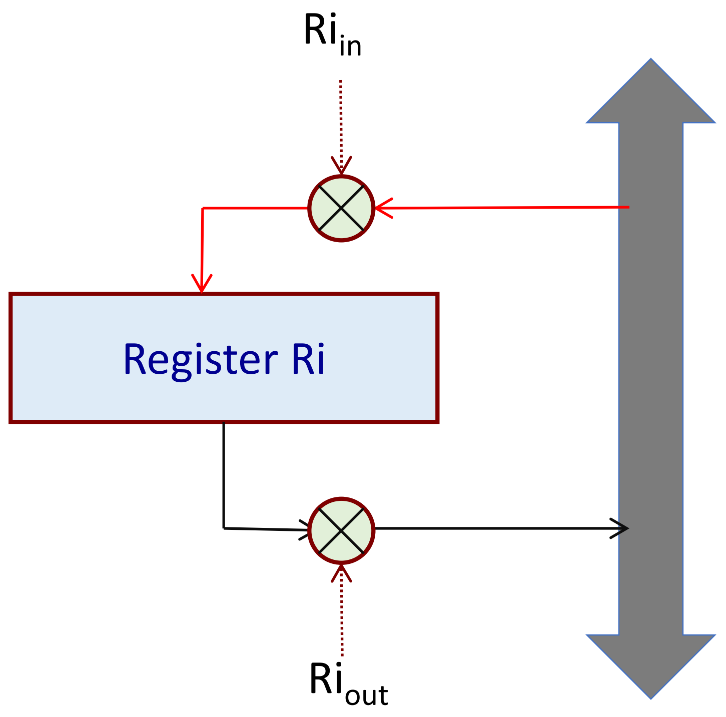

Bus A
Bus B
Incrementer
Bus C
PC
Register File
4
MUX
ALU
Instruction Decoder
IR
Data
MDR
Memory
MAR
Address

Organization of a Register

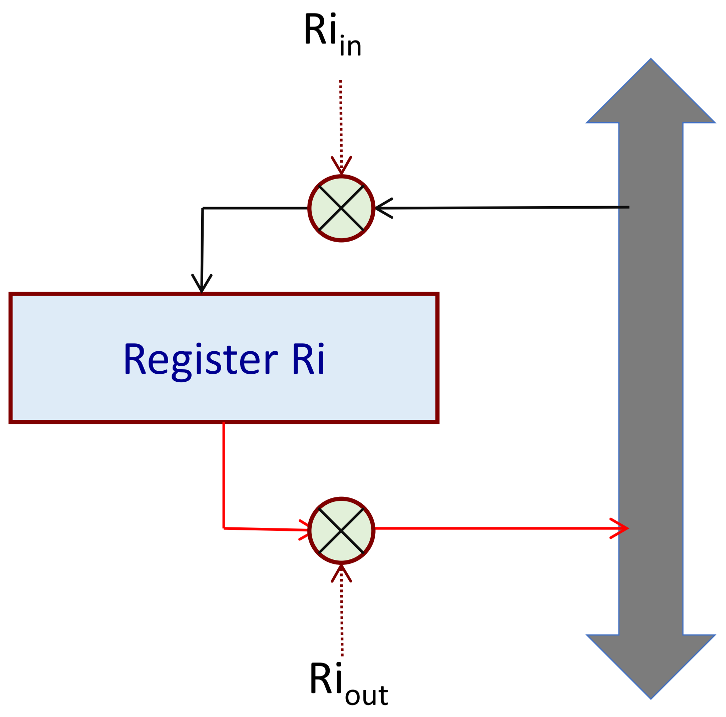
- A register is used for temporary storage of data (parallel-in, parallel-out, etc.).
- A register Ri typically has two control signals.
    - Ri_{in} : used to load the register with data from the bus.
    - Ri_{out} : used to place the data stored in the register on the bus.
- Input and output lines of the register Ri are connected to the bus via controlled switches.



- When ($Ri_{in} = 1$), the data available on bus is loaded into Ri.



- When (Ri_{out} = 1), the data from register Ri are placed on the bus.

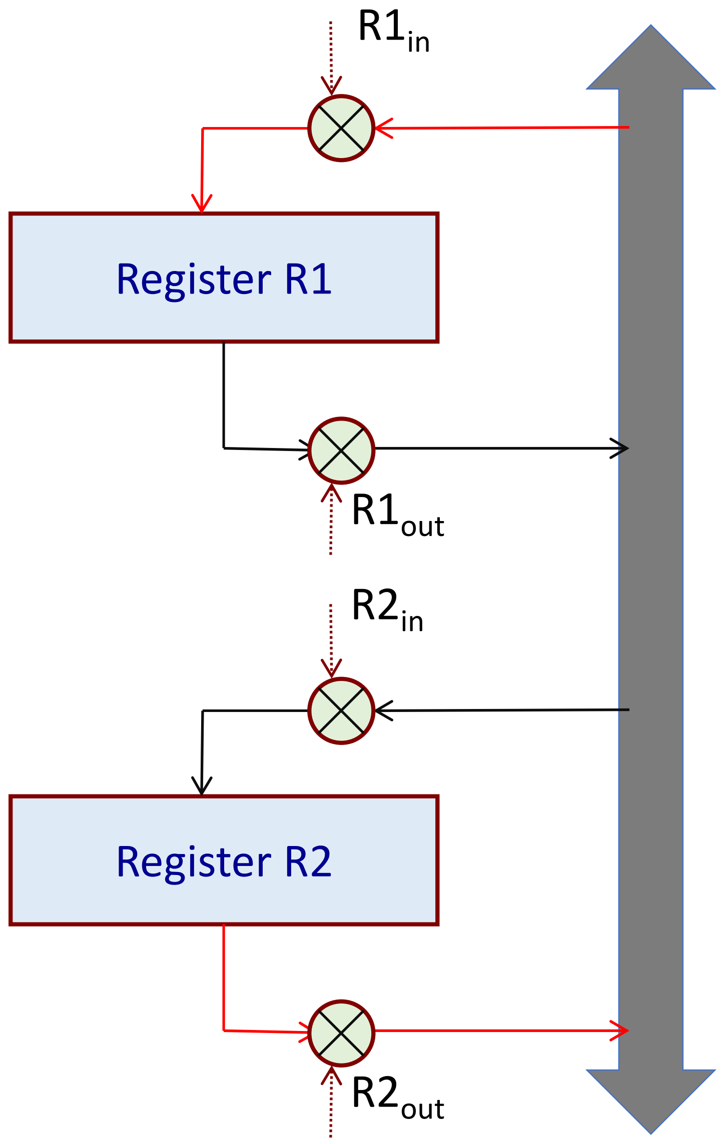


Register Transfer

MOVE R1, R2 // R1 ← R2

- Enable the output of R2 by setting $R2_{out} = 1$.
- Enable the input of register R1 by setting $R1_{in} = 1$.
- All operations are performed in synchronism with the processor clock.
  - The control signals are asserted at the start of the clock cycle.
  - After data transfer the control signals will return to 0.
- We write as $T1\colon\ R2_{out},\ R1_{in}$

Time Step **Control Signals**



ALU Operation

ADD R1, R2 // R1=R1 + R2

- Bring the two operands (R1 and R2) to the two inputs of the ALU.

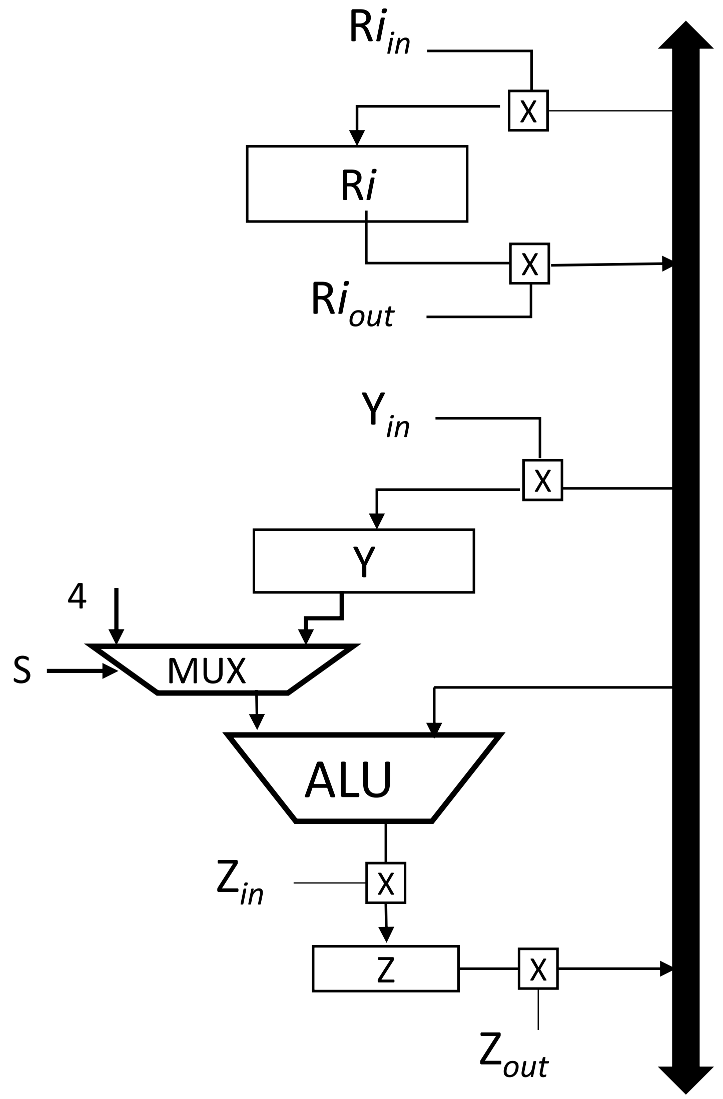
  One through Y (R1) and another (R2) directly from internal bus.

- Result is stored in Z and finally transferred to R1.

  T1: $R1_{out}$, Y_{in}

  T2: $R2_{out}$, *SelectY, ADD,* Z_{in}

  T3: Z_{out}, $R1_{in}$



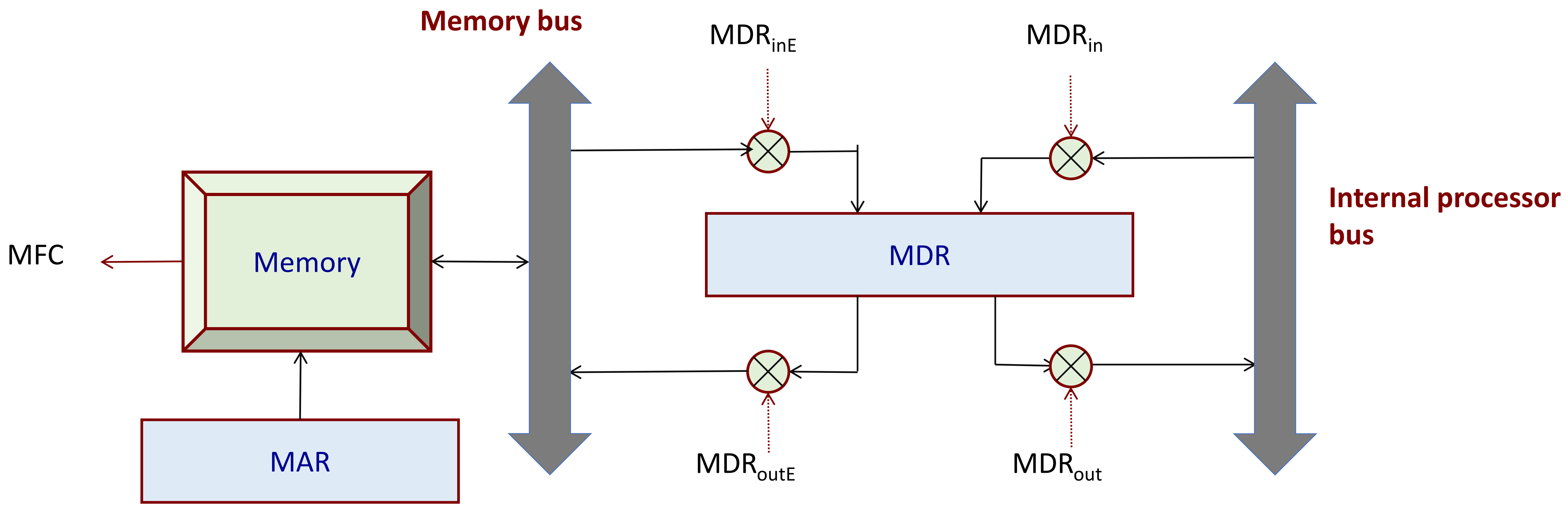
Fetching a Word from Memory

- The steps involved to fetch a word from memory:
  - The processor specifies the address of the memory location where the data or instruction is stored.
  - The processor requests a read operation.
  - The information to be fetched can either be an instruction or an operand of the instruction.
  - The data read is brought from the memory to MDR.
  - Then it can be transferred to the required register or ALU for further operation.

Storing a Word into Memory

- The steps involved to store a word into the memory:
  - The processor specifies the address of the memory location where the data is to be written.
  - The data to be written in loaded into MDR.
  - The processor requests a write operation.
  - The content of MDR will be written to the specified memory location.

Connecting MDR to Memory Bus and Internal Bus



- Memory read/write operation:
    - The address of memory location is transferred to MAR.
    - At the same time a read/write control signal is provided to indicate the operation.
    - For read the data from memory data bus comes to MDR by activating MDR_{inE}.
    - For write the data from MDR goes to memory data bus by activating the signal MDR_{outE}.

- When the processor sends a read request, it has to wait until the data is read from the memory and written into MDR.
- To accommodate the variability in response time, the process has to wait until it receives an indication from the memory that the read operation has been completed.
- A control signal called *Memory Function Complete* (MFC) is used for this purpose.
  - When this signal is 1, indicates that the content of the specified location is read and are available on the data line of the memory bus.
  - Then the data can be made available to MDR.

Fetch a word: MOVE R1, (R2)

1. MAR ← R2
2. Start a Read operation on the memory bus
3. Wait for the MFC response from the memory
4. Load MDR from the memory
5. R1 ← MDR

Control steps:

a) $R2_{out}$, MAR_{in}, Read
b) MDR_{inE}, WMFC
c) MDR_{out}, $R1_{in}$

Store a word: MOVE (R1), R2

1. MAR ← R1
2. MDR ← R2
3. Start a Write operation on the memory bus
4. Wait for the MFC response from the memory

Control steps:

a) $R1_{out}$, MAR_{in}

b) $R2_{out}$, MDR_{in}, Write

c) MDR_{outE}, WMFC

Execution of a Complete Instruction

ADD R1, R2   // R1 = R1 + R2

T1: PC_{out}, MAR_{in}, Read, Select4, ADD, Z_{in}

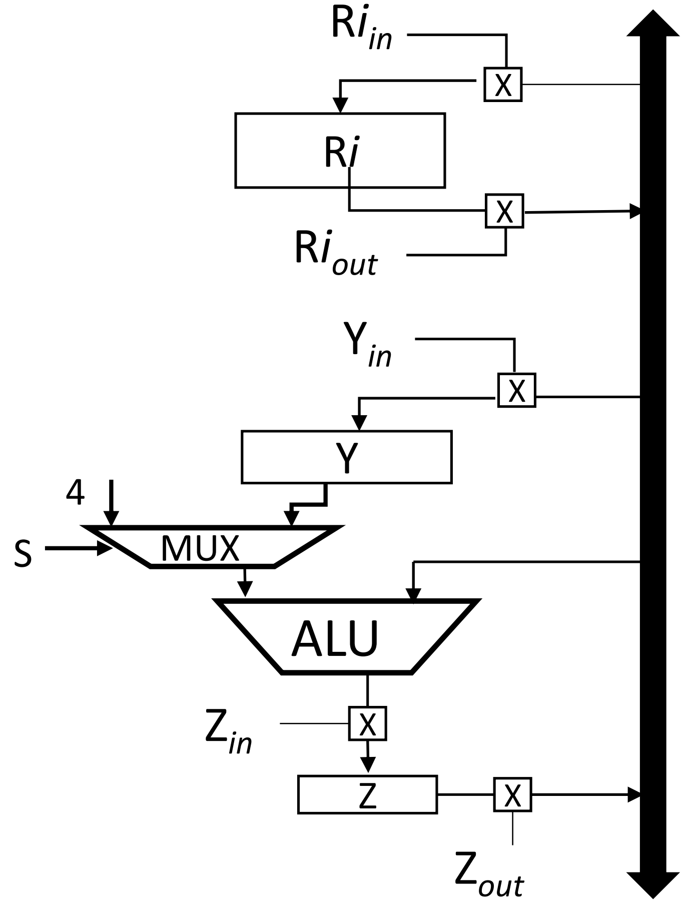
T2: Z_{out}, PC_{in}, Y_{in}, WMFC

T3: MDR_{out}, IR_{in}

T4: $R1_{out}$, Y_{in}, SelectY

T5: $R2_{out}$, ADD, Z_{in}

T6: Z_{out}, $R1_{in}$



Example for a Three Bus Organization

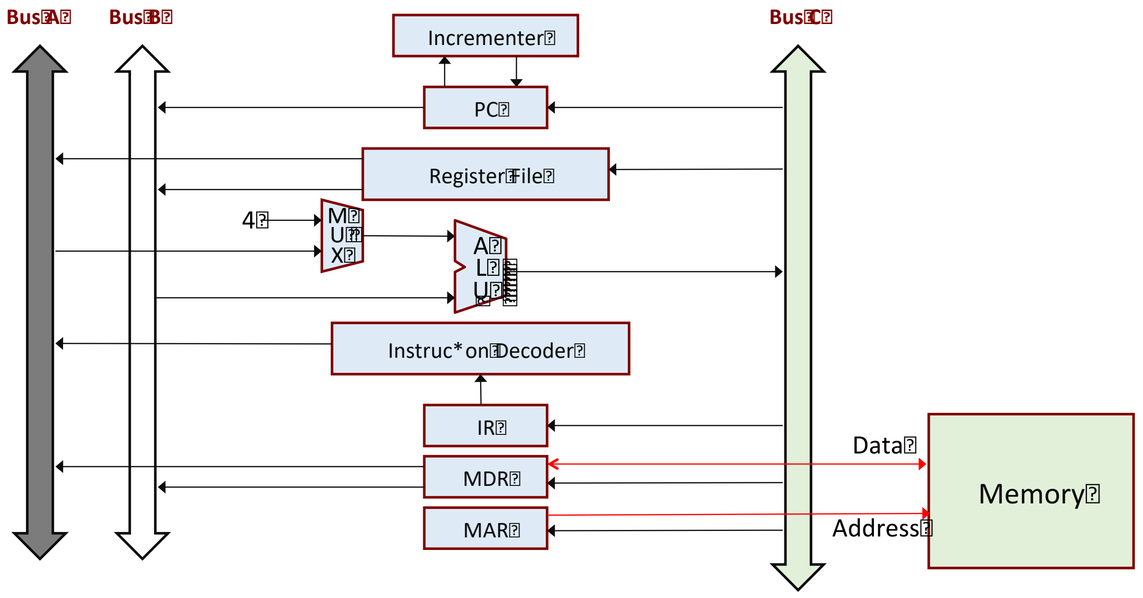
SUB R1, R2, R3 // R1 = R2 – R3

T1: PC_{out}, R = B, MAR_{in} , READ, IncPC

T2: WMFC

T3: MDR_{outB} , R = B, IR_{in}

T4: $R2_{outA}$, $R3_{outB}$, SelectA, SUB, $R1_{in}$, End



Micro-operations Examples

Introduction

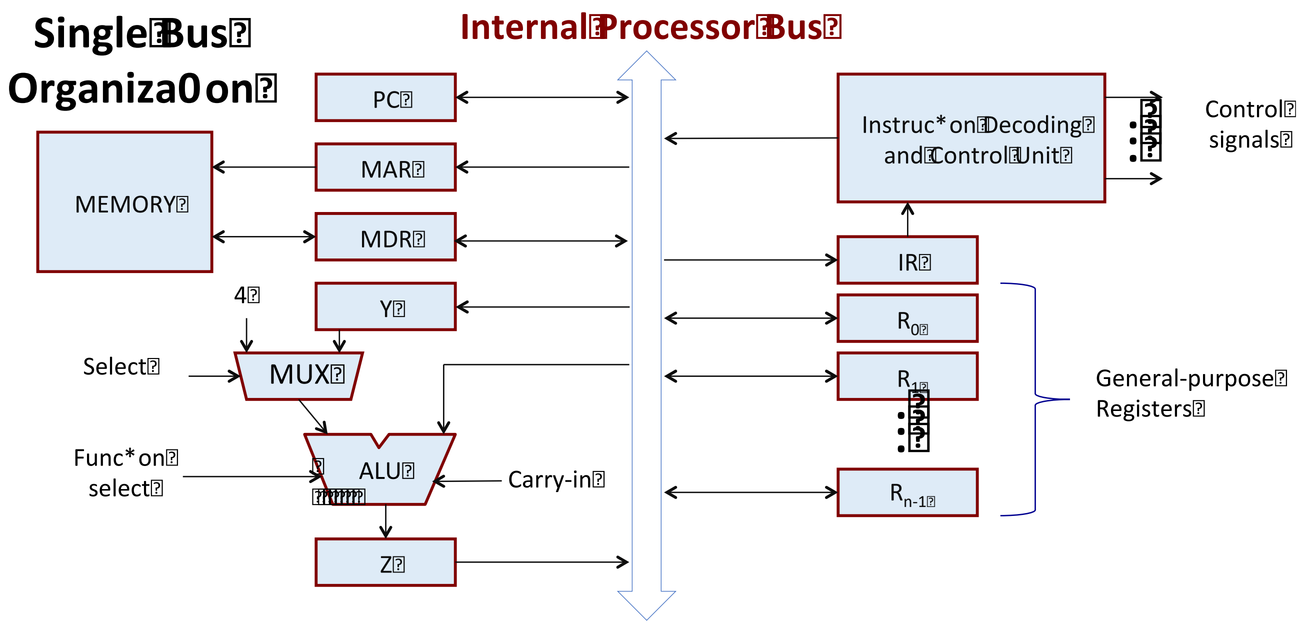
- We select a set of 12 instructions.
- Discuss the control signals required to execute these instructions on the single-bus processor architecture.

Various instructions: Control sequence

```
1.  ADD R1, R2          //  R1 = R1+R2
2.  ADD R1, LOCA        //  R1 = R1 + Mem[LOCA]
3.  LOAD R1, LOCA       //  R1 = Mem[LOCA]
4.  STORE LOCA, R1      //  Mem[LOCA] = R1
5.  MOVE R1, R2         //  R1 = R2
6.  MOVE R1, #10        //  R1 = 10
7.  BR LOCA             //  PC = LOCA
8.  BZ LOCA             //  PC = LOCA if Zero flag is set
9.  INC R1              //  R1 = R1 + 4
10. DEC R1              //  R1 = R1 – 4
11. CMP R1, R2          //  R1 – R2
12. HALT                // Machine Halt
```

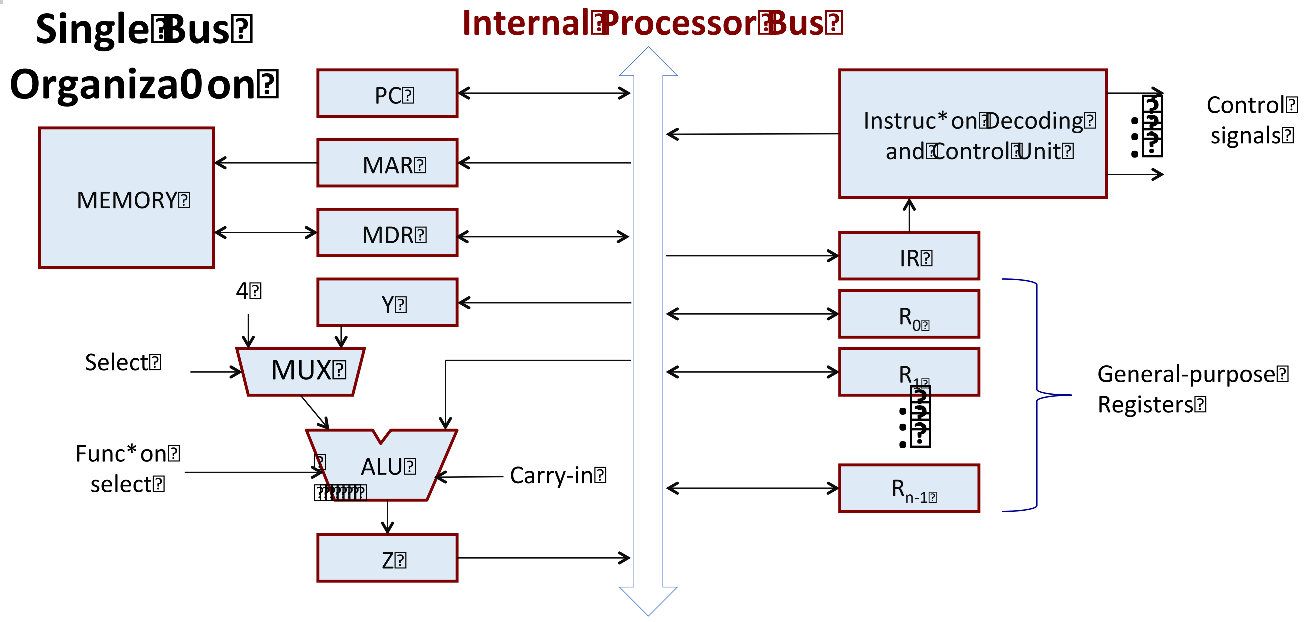
1. ADD R1, R2 (R1 = R1+R2)

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Y_{in}
5	$R2_{out}$, SelectY, Add, Z_{in}
6	Z_{out}, $R1_{in}$, End



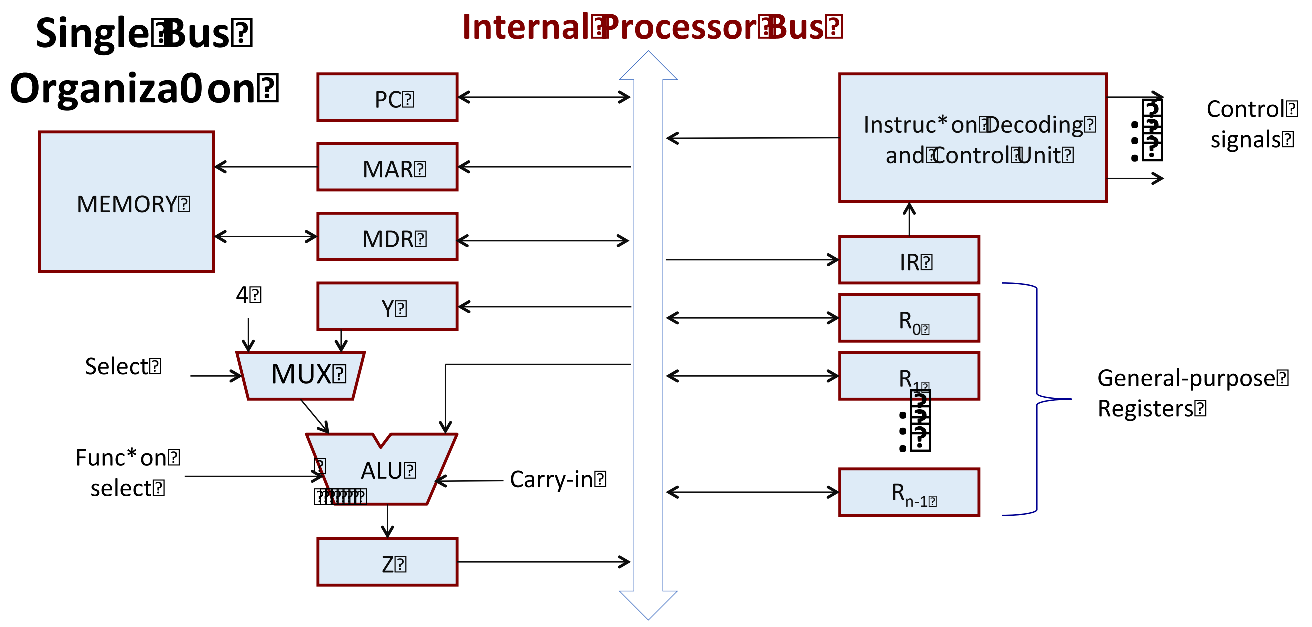
2. ADD R1, LOCA (R1 = R1 + Mem[LOCA])

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Address field of IRout, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End



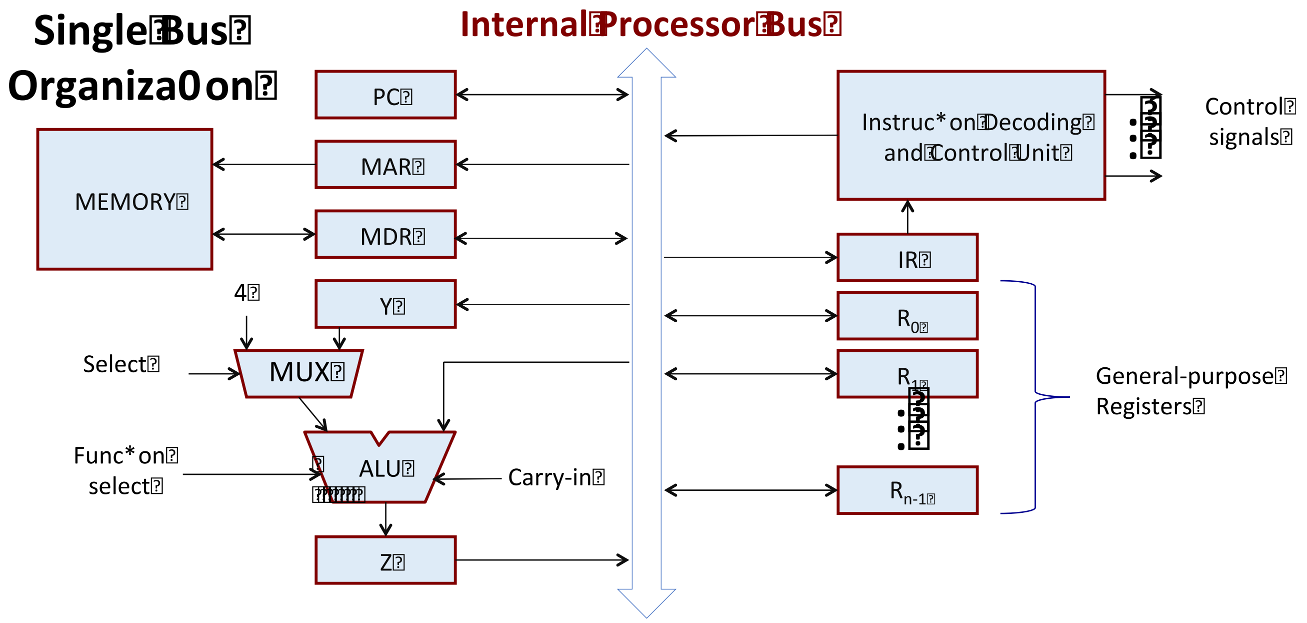
3. LOAD R1, LOCA (R1 = Mem[LOCA])

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Address field of IRout, MAR_{in}, Read
5	WMFC
6	MDR_{out}, $R1_{in}$, END



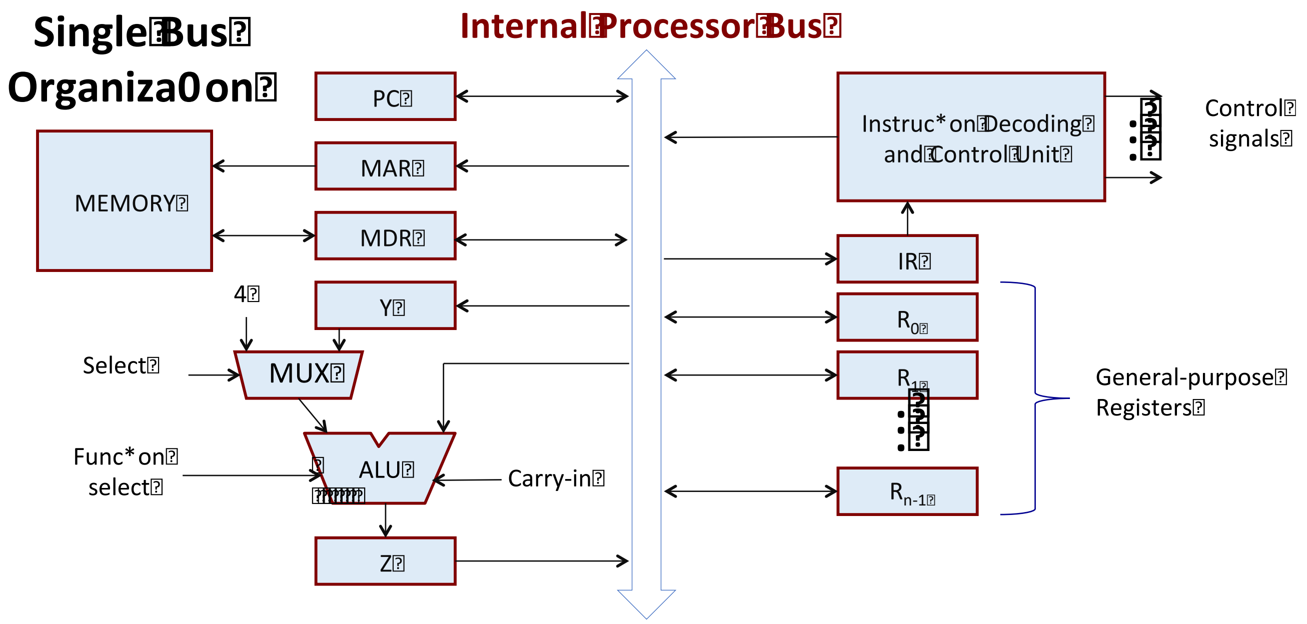
4. STORE LOCA, R1 (Mem[LOCA] = R1)

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Address field of IRout, MAR_{in}
5	$R1_{out}$, MDR_{in}, Write
6	MDR_{outE}, WMFC, End



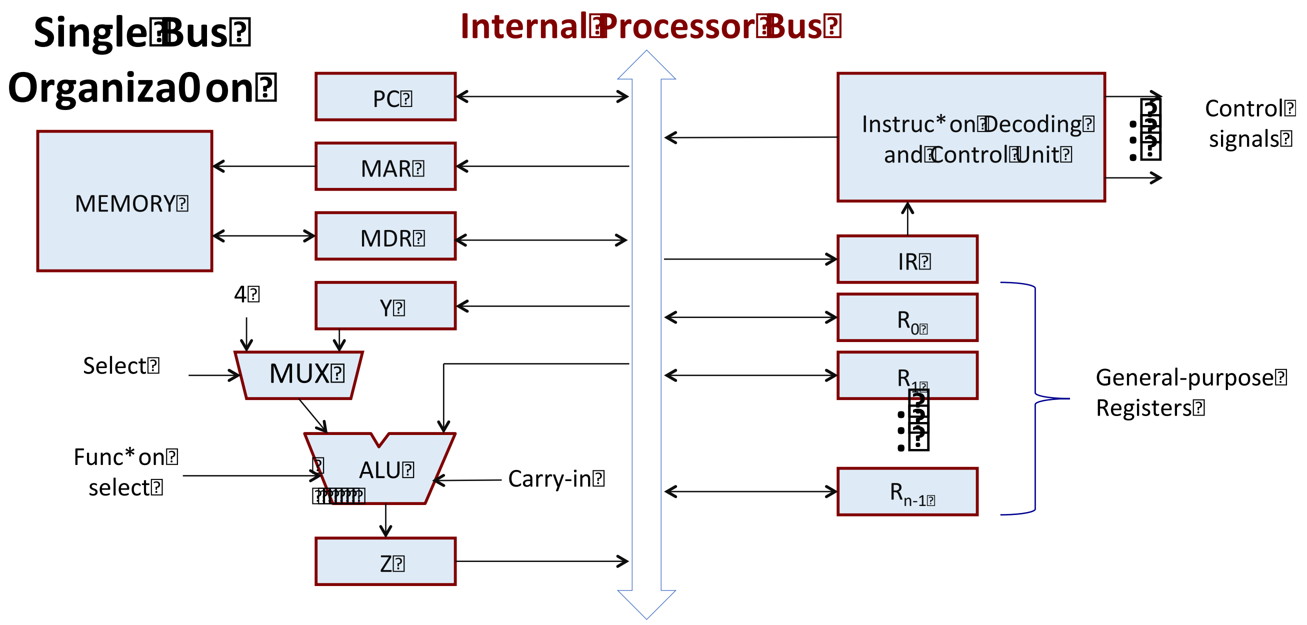
5. MOVE R1, R2 (R1 = R2)

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R2_{out}$, $R1_{in,}$ END



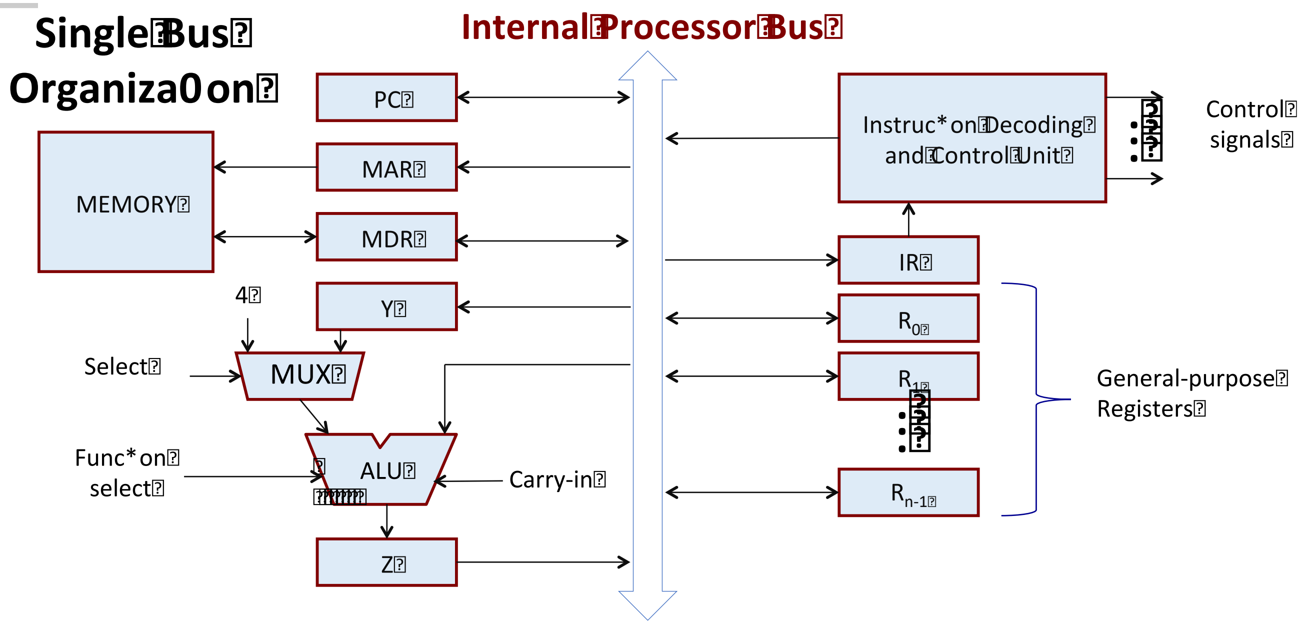
6. MOVE R1, #10 (R1 = 10)

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Immediate field of IR_{out}, $R1_{in}$, END



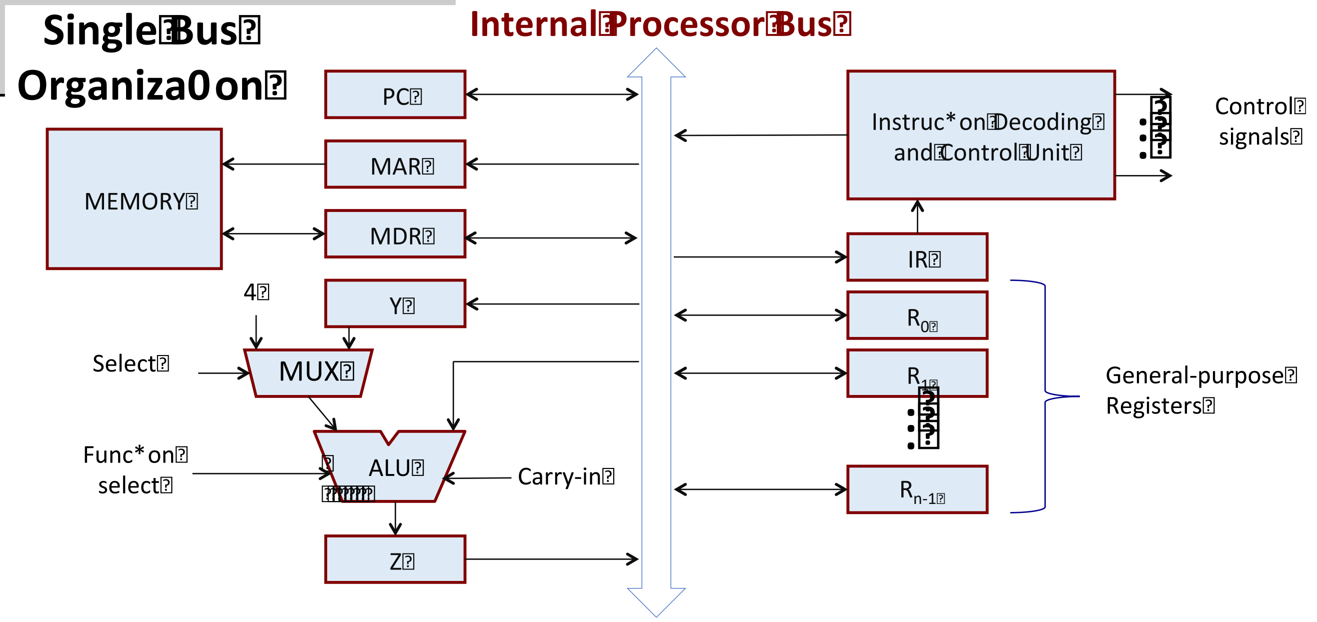
7. BRANCH Label (PC = PC + offset)

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
5	Z_{out}, PC_{in}, End



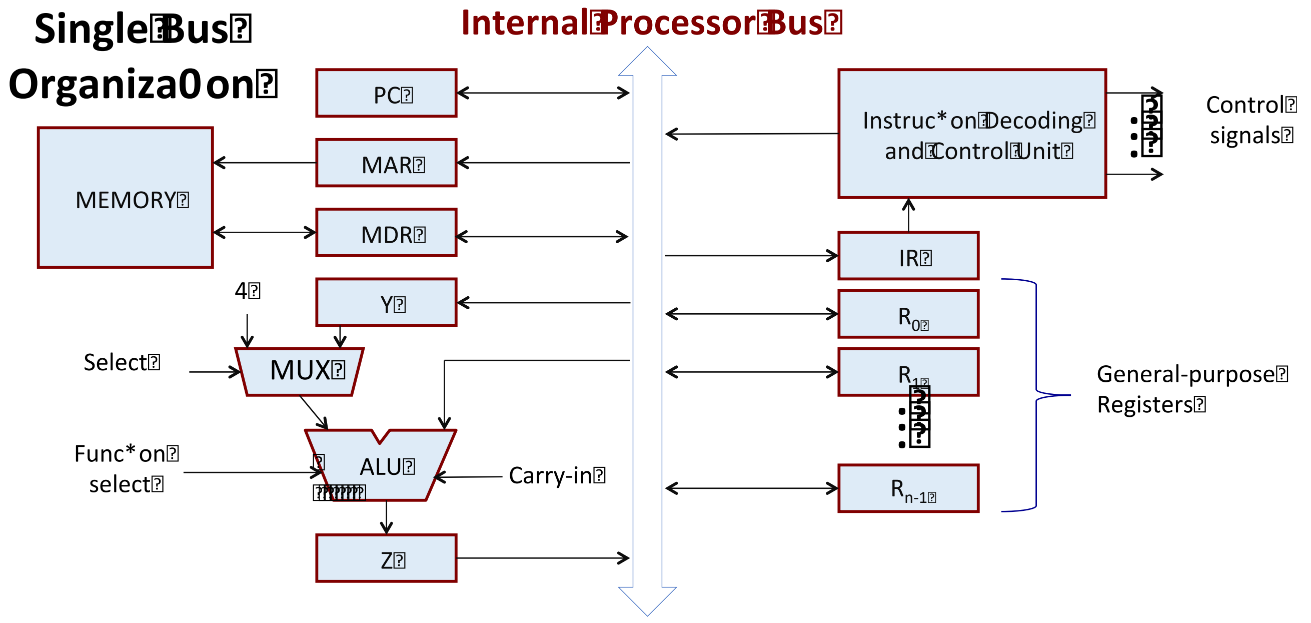
8. BZ Label (if Z=1 PC = PC + offset)

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}, If Z=0 then End
5	Z_{out}, PC_{in}, End



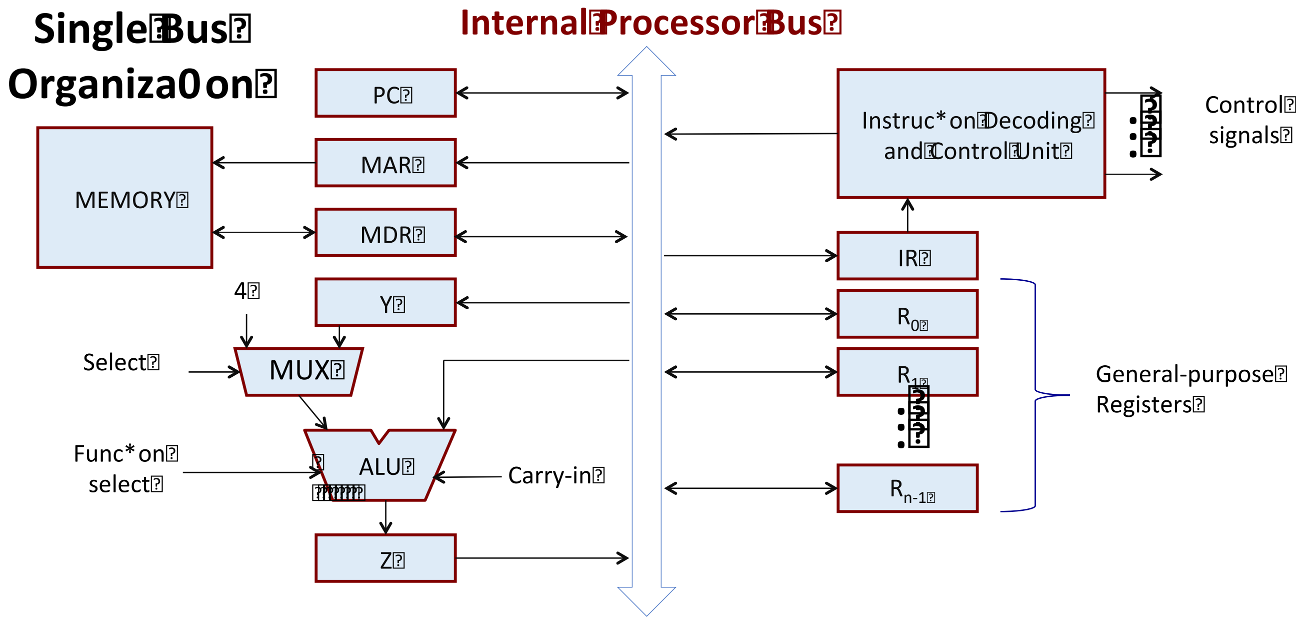
9. INC R1 (R1 = R1 + 4)

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Select4, Add, Z_{in}
5	Z_{out}, $R1_{in}$, End



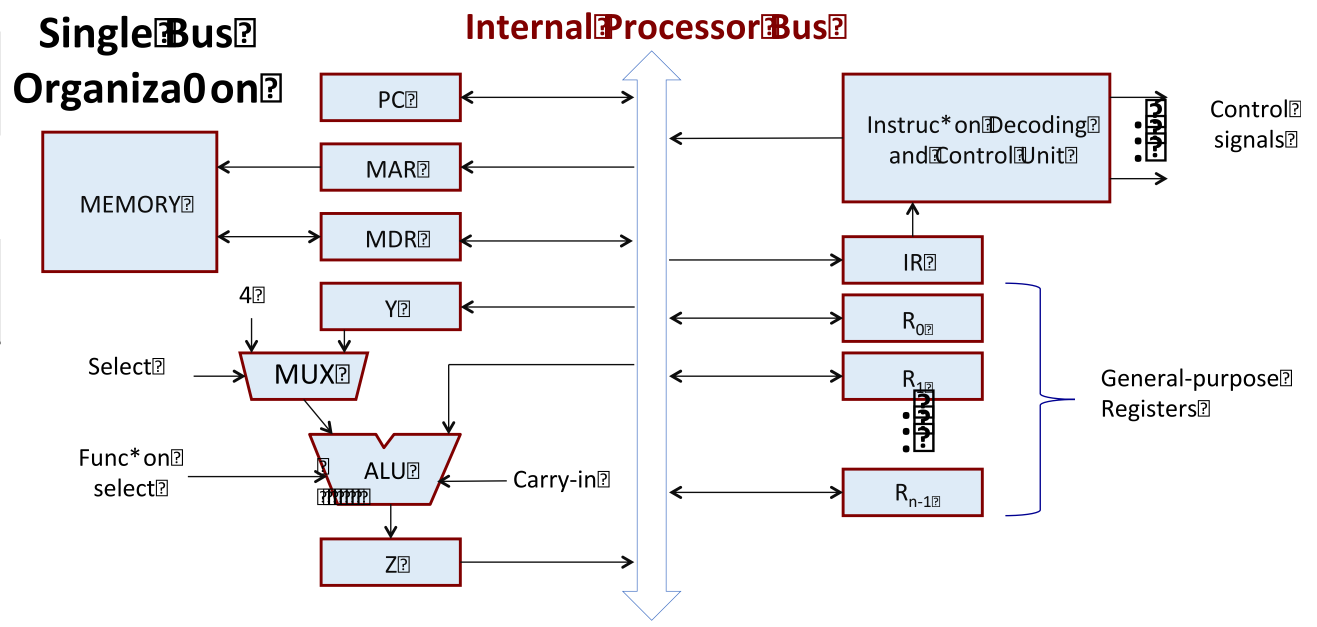
10. DEC R1 (R1 = R1 – 4)

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Select4, SUB, Z_{in}
5	Z_{out}, $R1_{in}$, End



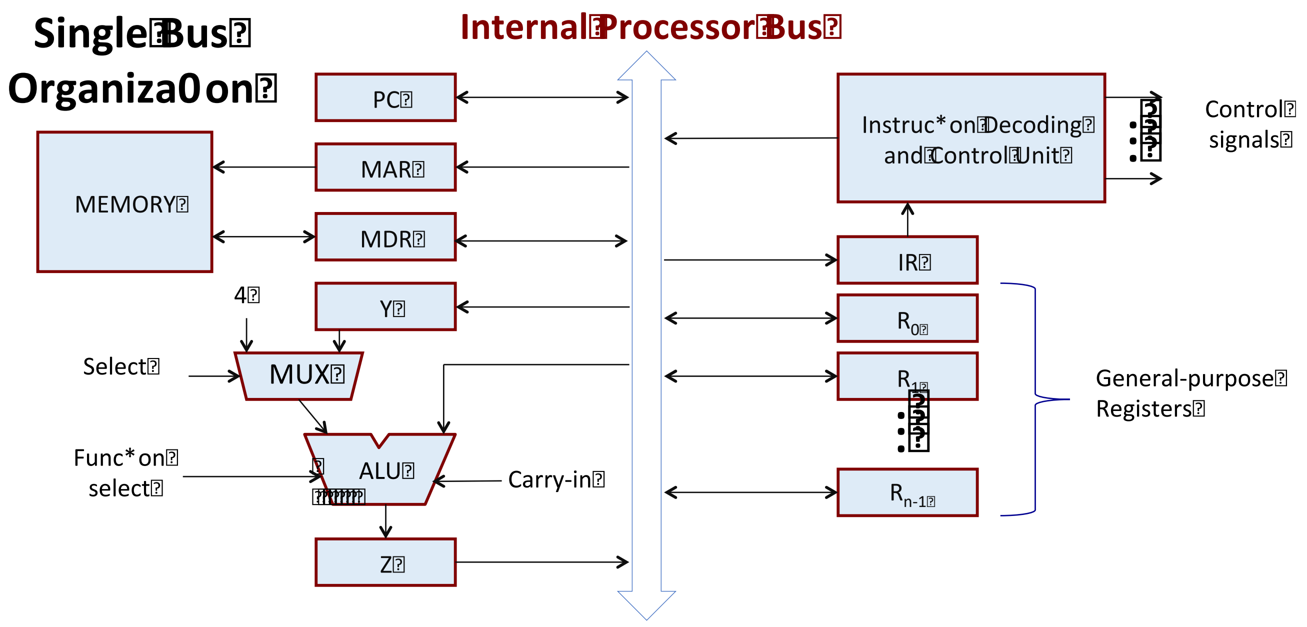
11. CMP R1, R2

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Y_{in}
5	$R2_{out}$, SelectY, Sub, Z_{in}, End



12. HALT

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	End

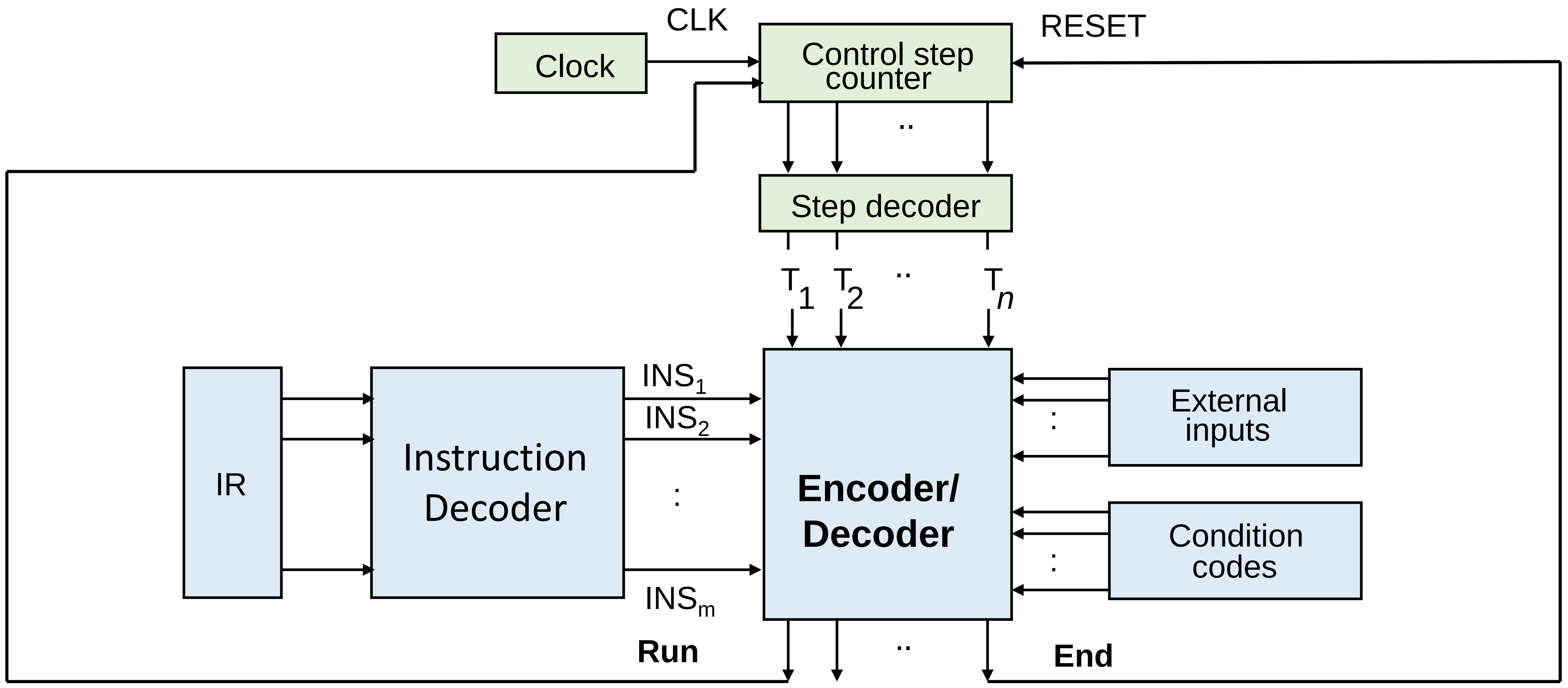


Hardwired and Microprogrammed Control Unit Design

Introduction

- To execute an instruction, the processor must generate control signals for the data path in proper sequence.
  - Example: ADD R1, R2
    a) $R1_{out}$, Y_{in}, SelectY
    b) $R2_{out}$, ADD, Z_{in}
    c) Z_{out}, $R1_{in}$
- Two alternate approaches:
  1. Hardwired control unit design
  2. Microprogrammed control unit design

Hardwired Control unit



Control signals

Sequence of control signals for ADD R1, LOCA

Steps	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Address field of IR_{out}, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Hardwired Control Unit Design

- Assumption:
  - Each step in this sequence is completed in one clock cycle.
- A counter is used to keep track of the time step.
- The control signals are determined by the following information:
  - Content of control step counter
  - Content of instruction register
  - Content of conditional code flags
  - External input signals such as MFC (Memory Function Complete)

- The encoder/decoder circuit is a combinational circuit which generates control signals depending on the inputs provided.
- The step decoder generates separate signal line for each step in the control sequence (T_1, T_2, T_3, etc.).
  - Depending on maximum steps required for an instruction, the step decoder is designed.
  - If a maximum of 10 steps are required, then a 4 x 16 step decoder is used.
- Among the total set of instructions, the instruction decoder is used to select one of them. (That particular line will be 1 and rest will be 0).
  - If a maximum of 100 instructions are present in the ISA then a 7 x 128 instruction decoder is used.

- At every clock cycle the RUN signal is used to increment the counter by one.
  - When RUN is 0 the counter stops counting.
  - This signal is needed when WMFC is issued.
- END signal starts a new instruction.
  - It resets the control step counter to its starting value.
- The sequence of operations carried out by the control unit is determined by the wiring of the logic elements and hence it is named *hardwired*.
- This approach of control unit design is fast but limited to the complexity of instruction set that is implemented.

Generation of Control Signals

	ADD R1, R2
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Y_{in}
5	$R2_{out}$, SelectY, Add, Z_{in}
6	Z_{out}, $R1_{in}$, End

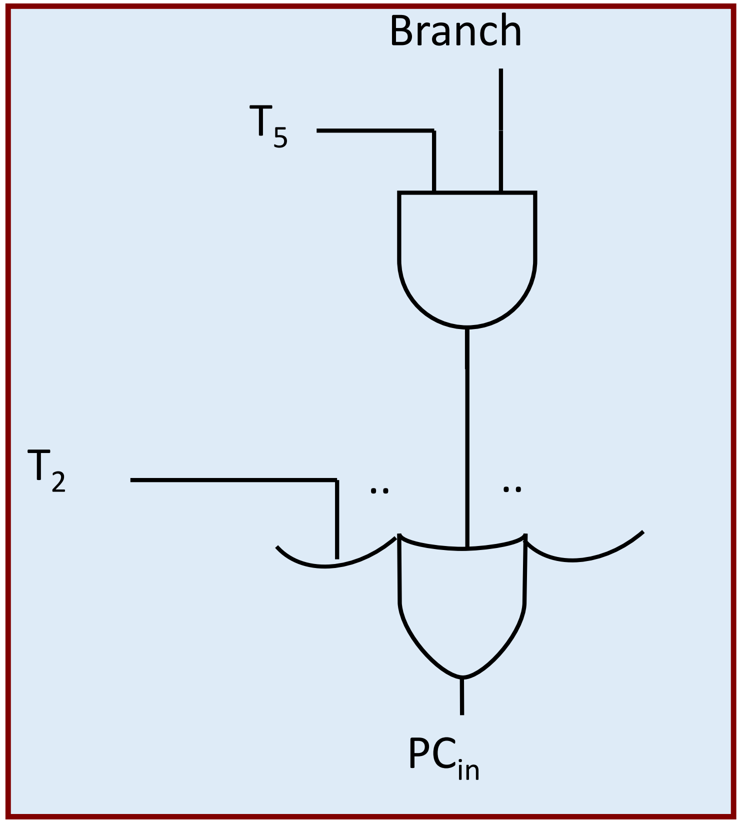
	ADD R1, LOCA
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Address field of IRout, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

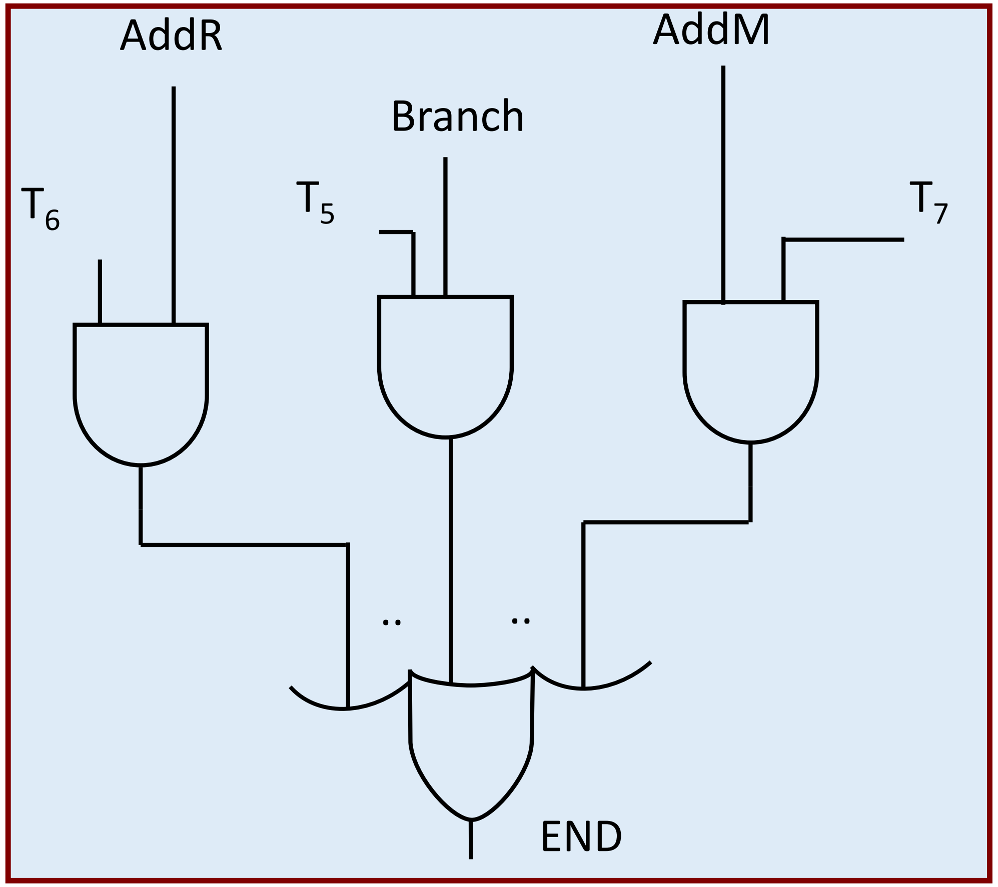
	BRANCH Label
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Generation of PC_{in} and END

$PC_{in} = T_2 + T_5.Branch$

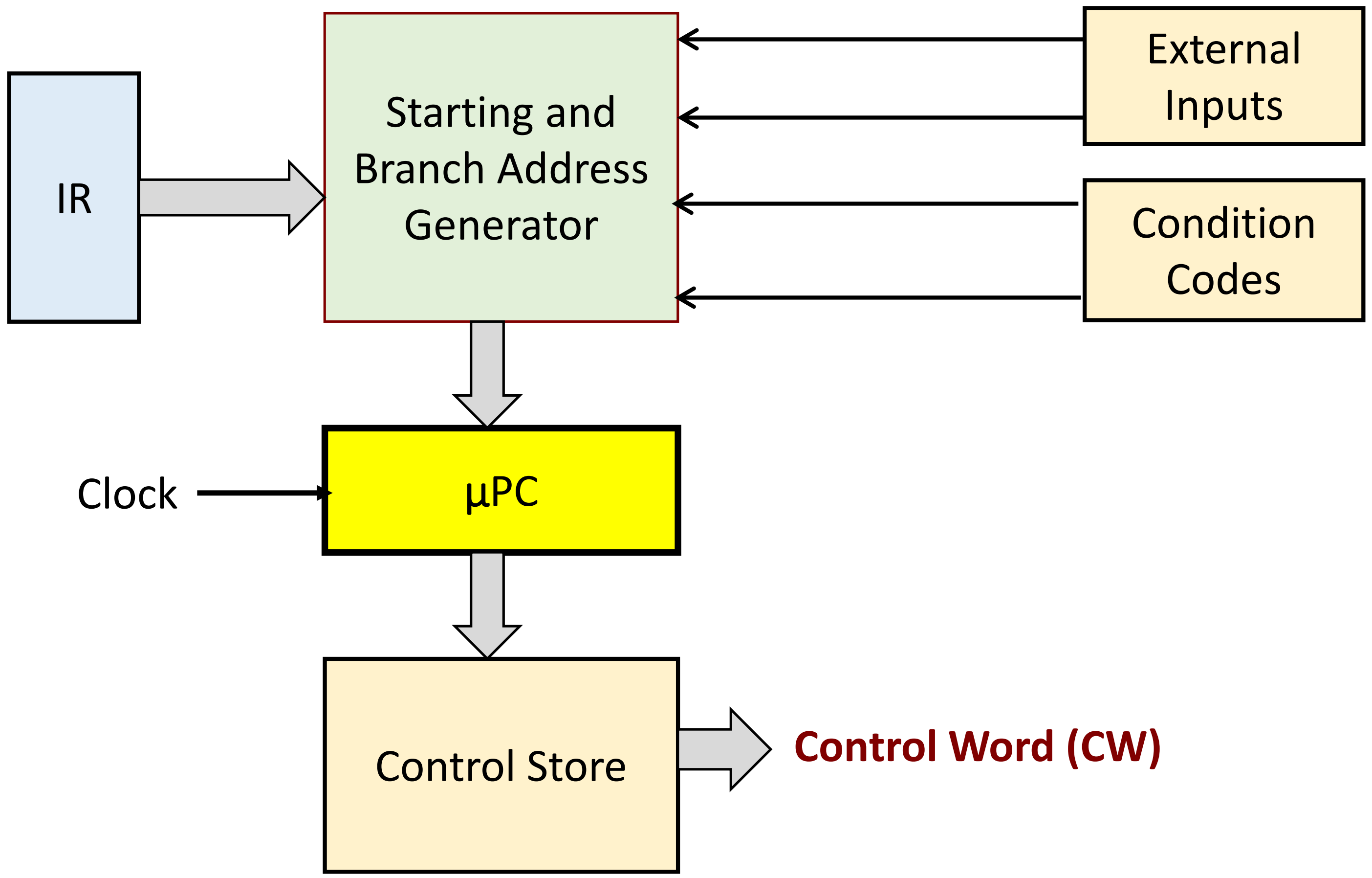
$END = T_6.ADDR + T_5.Branch + T_7.AddM$



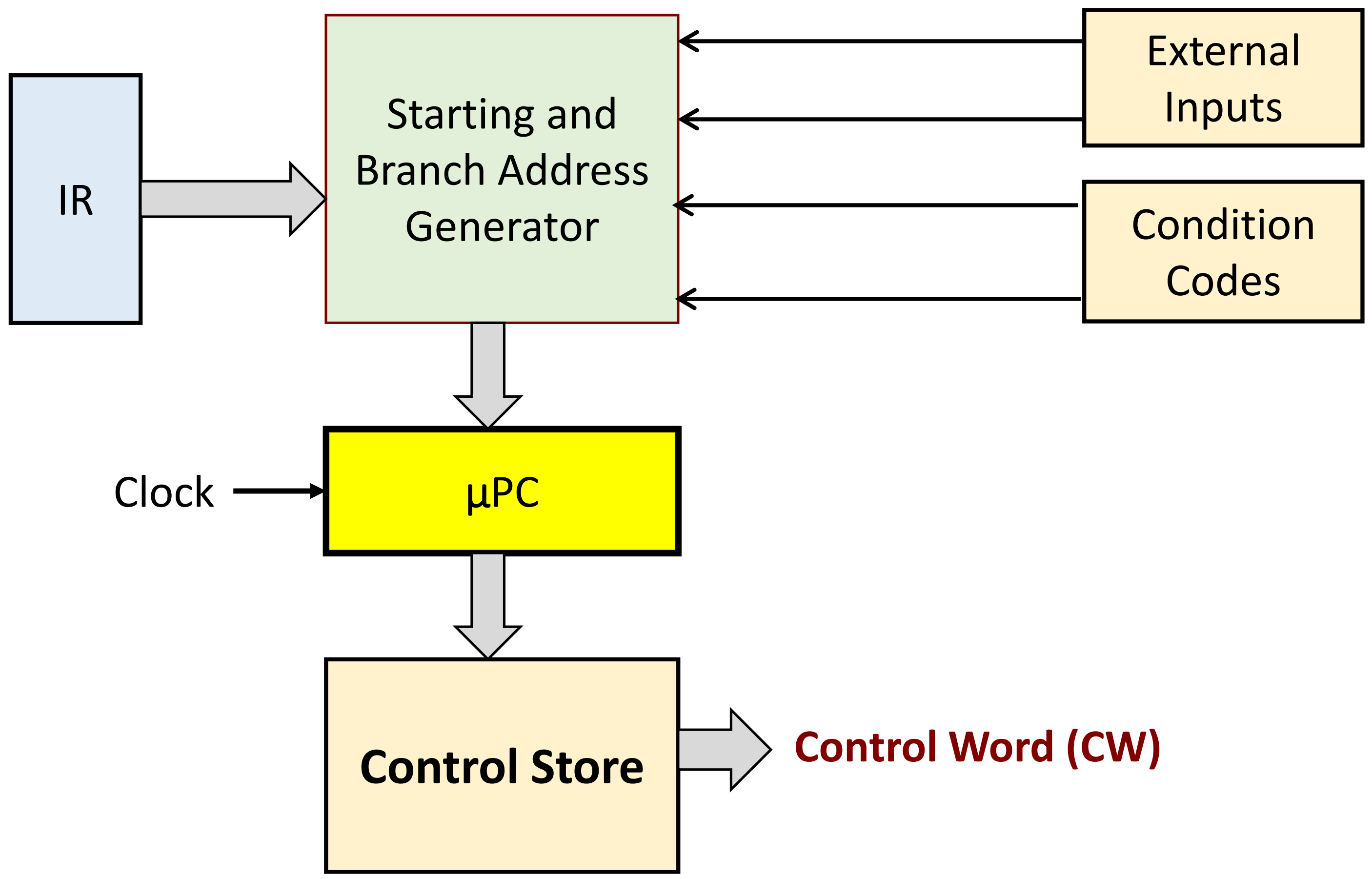


Microprogrammed Control Unit Design

- Control signals are generated by a program similar to machine language program.
- A *Control Store* (CS) stores the microroutines for all instructions of an ISA.
- The sequence of steps corresponding to the control sequence of a machine instruction is the *microroutine*.
- Each sequence of steps is a *control word* (CW) whose individual bits represent the various control signals.
- Individual control words in a microroutine are called *microinstructions*.


Starting and Branch Address Generator
IR
External Inputs
Condition Codes
Clock
μPC
Control Store
Control Word (CW)

- Control-unit generates the control signals for an instruction by sequentially reading CWs of corresponding microroutine from CS.
- The *μPC* is used to read CWs sequentially from CS.
- Every time a new instruction is loaded into IR, output of *Starting Address Generator* is loaded into μPC.
- Then, μPC is automatically incremented by clock causing successive microinstructions to be read from *Control Store*.



Control Store Contents for an Example Instruction

	ADD R1, R2
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R1_{out}$, Y_{in}
5	$R2_{out}$, SelectY, Add, Z_{in}
6	Z_{out}, $R1_{in}$, End

Control Store for "ADD R1, R2"

Micro-instr.	…	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select4	SelectY	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R2_{out}$	WMFC	End	…
1	0	0	1	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
3	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0

Control Store Contents for Another Instruction

	BRANCH Label
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
5	Z_{out}, PC_{in}, End

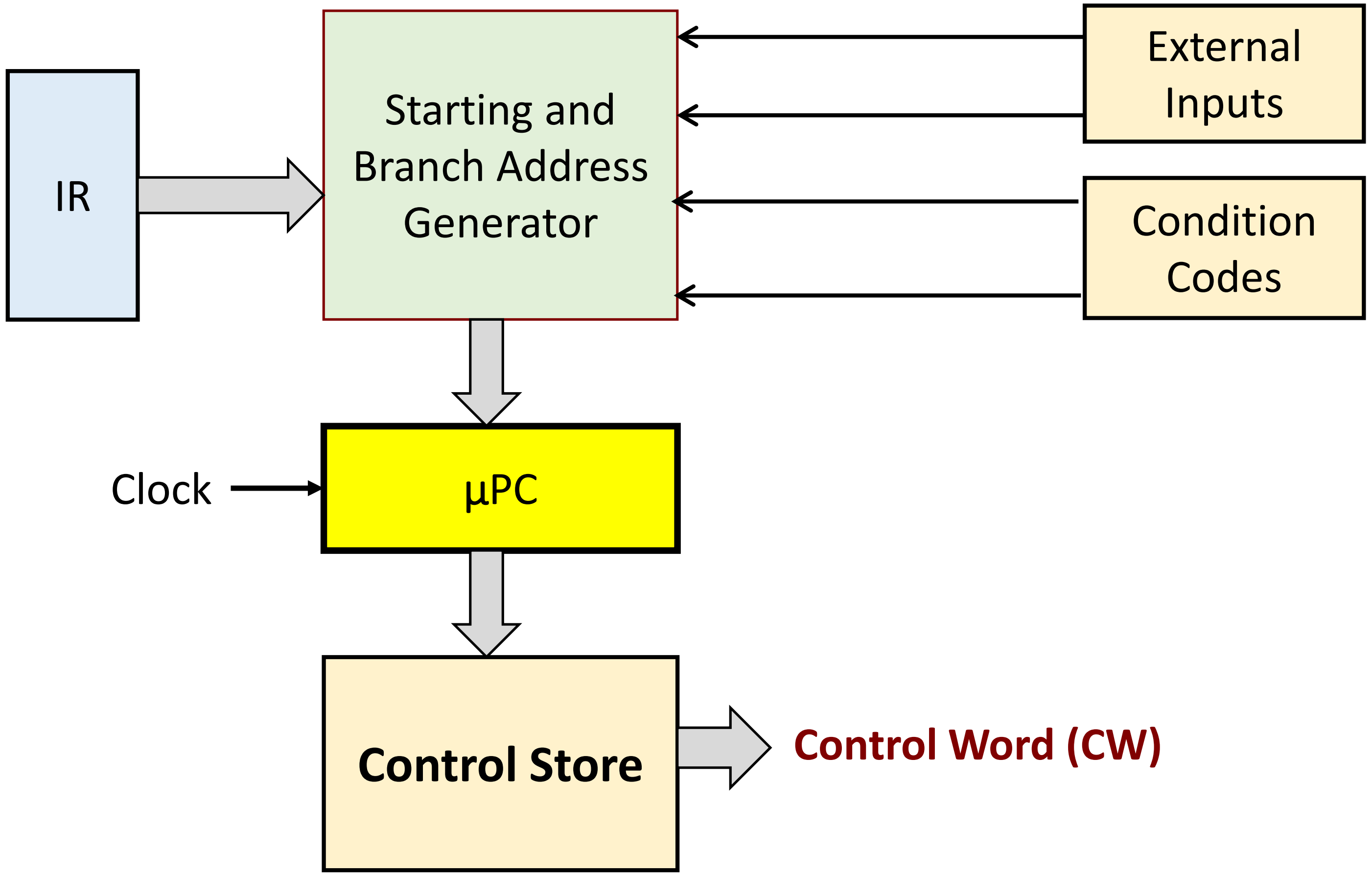
Control Store for “BRANCH Label”

Micro-instr.	…	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select4	SelectY	Add	Z_{in}	Z_{out}	IR_{out}	WMFC	End	…
1	0	0	1	1	1	0	0	0	1	0	1	1	0	0	0	0	0
2	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0
3	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0
5	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0

Using Conditional Branching

- The previous organization cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.
- Possible solution:
  - Use conditional branch *microinstruction*.

Address	Microinstruction
0	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}
1	Z_{out} , PC_{in} , Y_{in} , WMF C
2	MDR_{out} , IR_{in}
3	Branch to starting address of appropriate microroutine
. . . .	
25	If Z=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out} , SelectY, Add, Z_{in}
27	Z_{out} , PC_{in} , End


IR
Starting and Branch Address Generator
External Inputs
Condition Codes
Clock
μPC
Control Store
Control Word (CW)

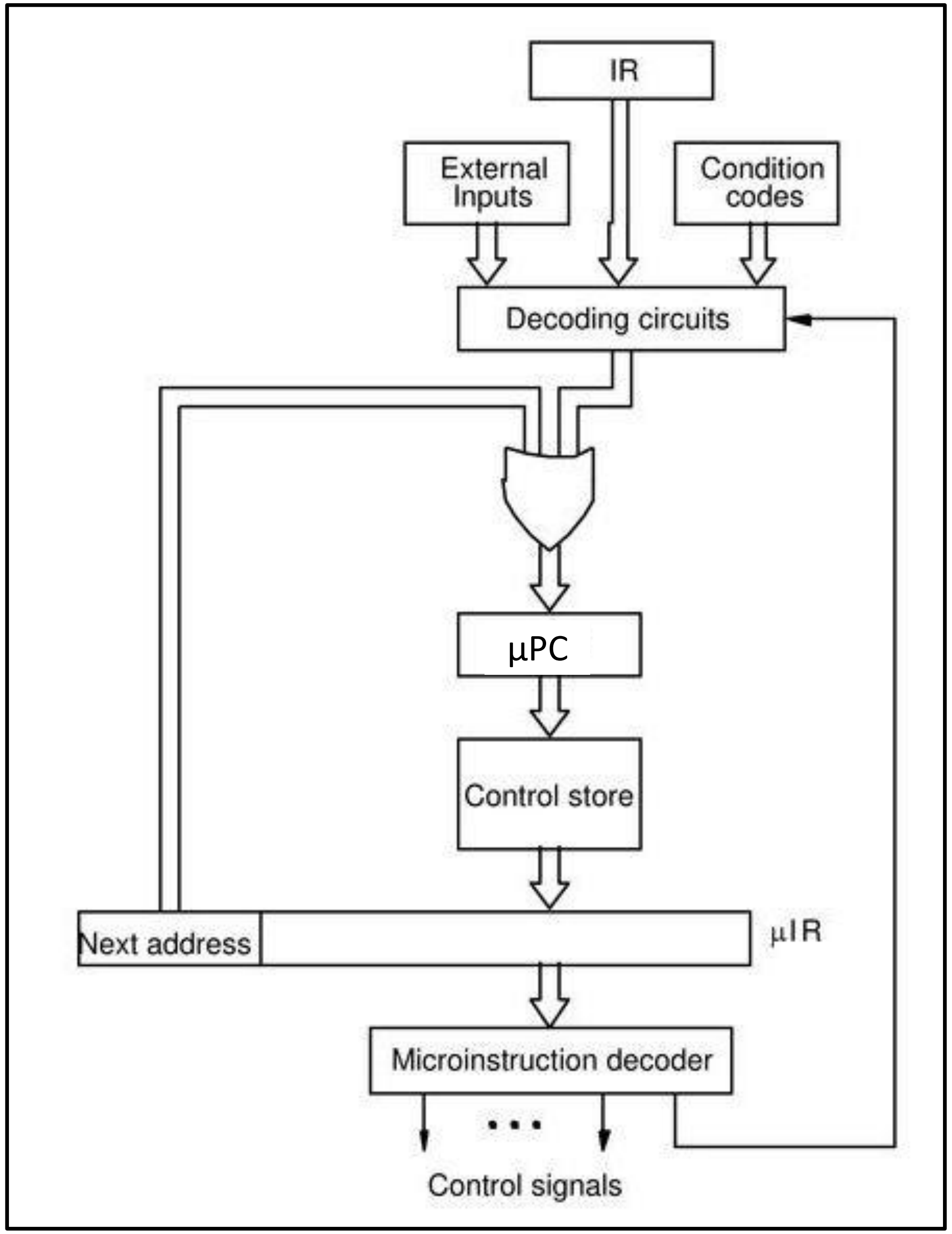
Microprogram Sequencing

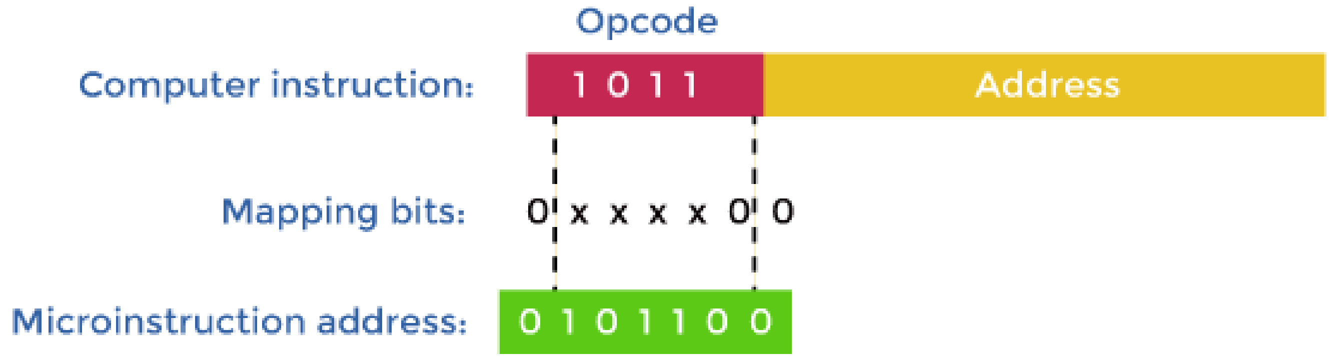
- If all microprograms require only straightforward sequential execution of microinstructions except for branches, letting a μPC govern the sequencing would be efficient.
- However, two disadvantages:
  - Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control store.
  - Longer execution time because it takes more time to carry out the required branches.

Microinstructions with Next-Address Field

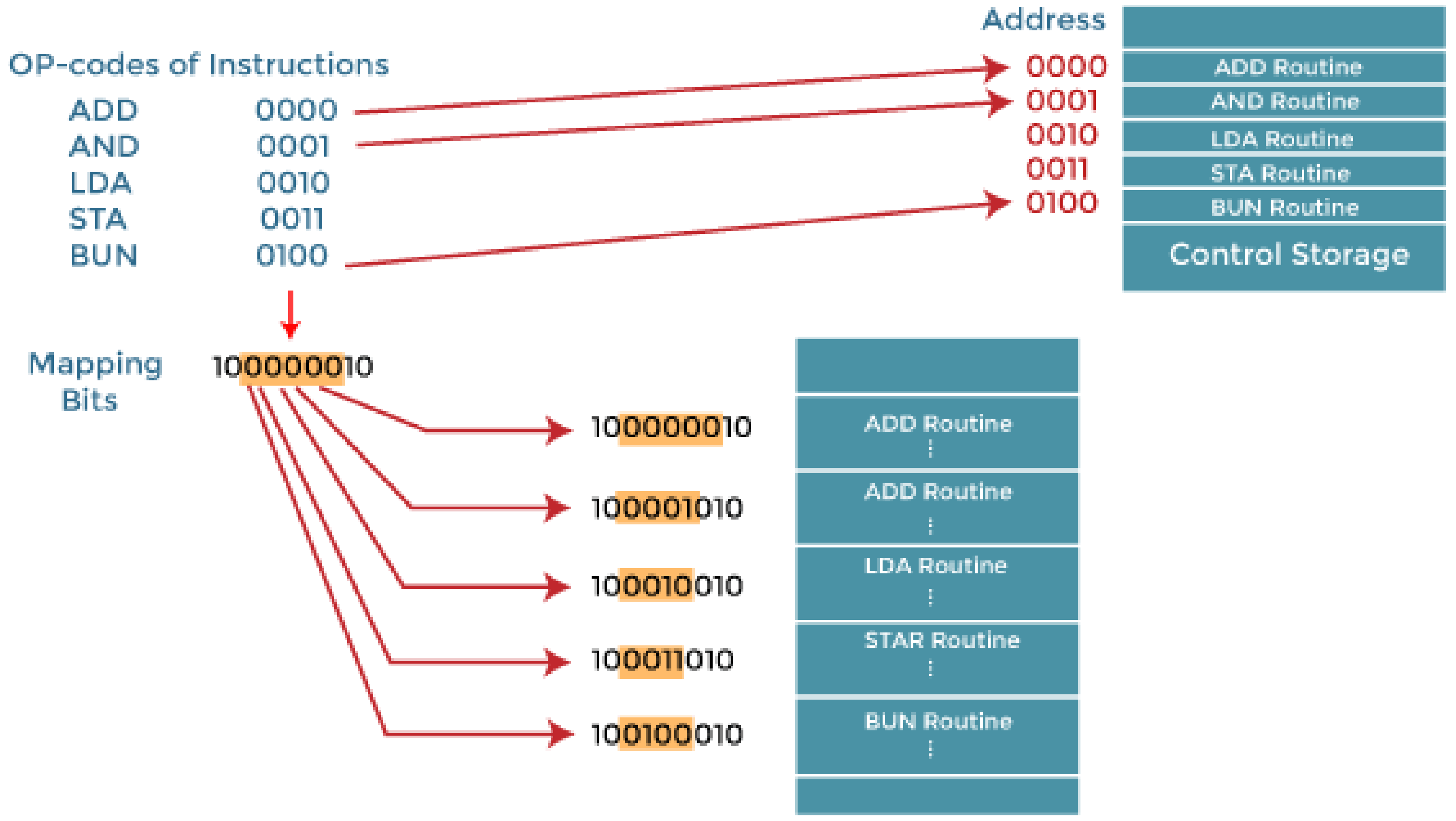
- The microprogram requires several branch microinstructions, which perform no useful operation in the datapath.
- An alternative approach:
  - Include an address field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched.

- Pros: separate branch microinstructions are virtually eliminated; few limitations in assigning addresses to microinstructions.
- Cons: additional bits required for the address field.





Mapping from instruction code to microinstruction address


OP-codes of Instructions
ADD 0000
AND 0001
LDA 0010
STA 0011
BUN 0100
Address
0000
0001
0010
0011
0100
ADD Routine
AND Routine
LDA Routine
STA Routine
BUN Routine
Control Storage
Mapping Bits
100000010
100000010
100001010
100010010
100011010
100100010
ADD Routine
ADD Routine
LDA Routine
STAR Routine
BUN Routine

Horizontal versus Vertical Microinstruction Encoding

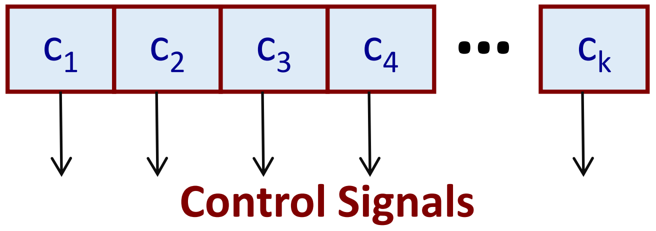
- Broadly there are two alternate schemes to code the control signals in the control memory.

  a) **Horizontal Micro-instruction Encoding**
     - Each control signal is represented by a bit in the micro-instruction.
     - Fewer control store words, with more bits per word.

  b) **Vertical Micro-instruction Encoding**
     - Each control word represents a single micro-instruction in encoded form.
     - k-bit control word can support up to 2^k micro-instructions.
     - More control store words, with fewer bits per word.

- There can be a tradeoff between horizontal and vertical micro-instruction encoding.
    - Sometimes referred to as **Diagonal Micro-instruction Encoding**.
    - The control signals are grouped into sets S_1, S_2, etc., such that the control signals within a set are mutually exclusive.
- Summary:
    - Horizontal encoding supports unlimited parallelism among micro-instructions.
    - Vertical encoding supports strictly sequential execution of micro-instructions.
    - Diagonal encoding does not sacrifice the required level of parallelism, but uses less number of bits per control word as compared to horizontal encoding.

(a) Horizontal Micro-instruction Encoding



- Suppose that there are *k* control signals: $c_1, c_2, \ldots, c_k$.
- In horizontal encoding, every control word stored in control memory (CM) consists of *k* bits, one bit for every control signal.
- Several bits in a control word can be 1:
    - Parallel activation of several micro-operations in a single time step.

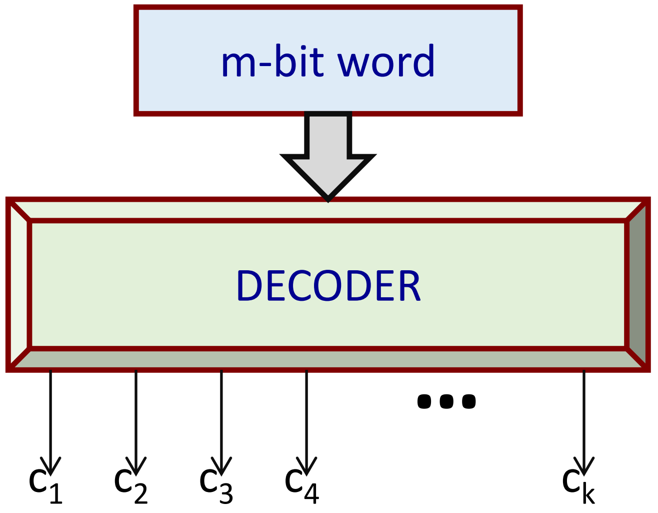
0	1	0	1	1	0

→ c_2, c_4 and c_5 are activated together

- Advantage:
  - Unlimited parallelism is possible in the activation of the micro-operations.
- Disadvantage:
  - Size of the control memory is large (word size is much longer).
  - Cost of implementation is higher.

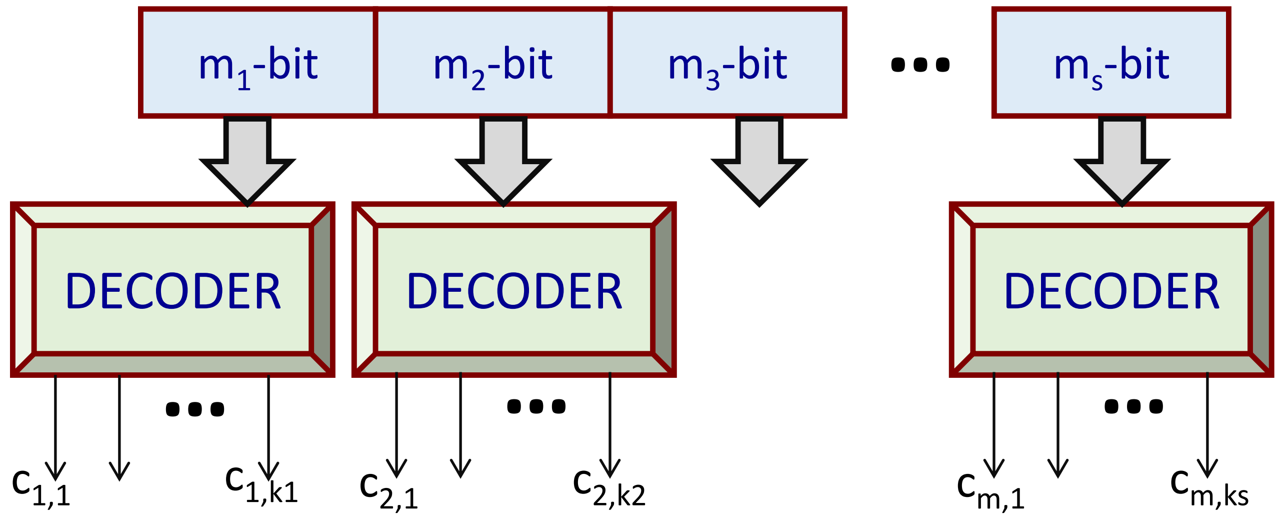
(b) Vertical Micro-instruction Encoding

- Again consider that there are k control signals: c_1, c_2, …, c_k.
- We encode the control signals in an m-bit word in the control memory, where $k \leq 2^m$.
- Depending on the m-bit control word, exactly one control signal will be activated (= 1), while all others will remain de-activated (= 0).
    - At most one control signal can be activated in a time step.



- Advantage:
    - Requires much smaller word size in control memory.
    - Low cost of implementation.
- Disadvantage:
    - More than one control signals cannot be activated at a time.
    - Requires sequential activation of the control signals, and hence more number of time steps.

(c) Diagonal Micro-instruction Encoding



- Suppose we group the set of *k* control signals into *s* groups, containing $k_1, k_2, \ldots, k_s$ signals.
- We encode the control signals in groups as shown, where $k_i \leq 2^{mi}$.
  - Within a group, at most one control signal can be activated in a time step.
  - Parallelism across groups is allowed.

- Advantages:
  - Maximum parallelism as required by the micro-programs can be supported.
  - Word size of control memory is less than that for horizontal encoding.
  - Used in practice.
- Disadvantages:
  - Multiple decoders (though smaller in sizes) are required.

Example 1

- Suppose there are 100 control signals in a processor data path.

  a) For horizontal encoding, control word size = 100 bits.

  b) For vertical encoding, control word size = $\lceil \log_2 100 \rceil$ = 7 bits.

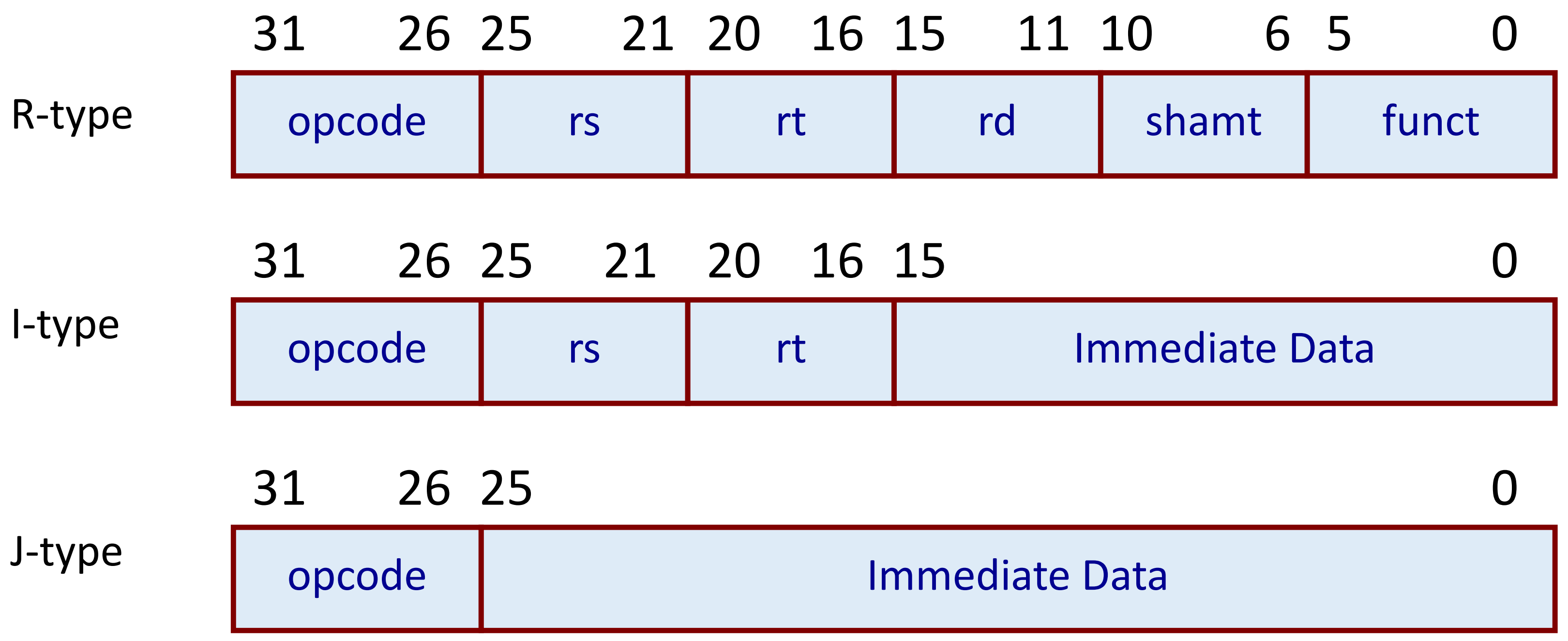
  c) For diagonal encoding, suppose after analysis of the micro-programs, we divide the control signals into 5 groups, containing 25, 15, 40, 5 and 15 control signals respectively.

  - We have: $m_1 = 5$, $m_2 = 4$, $m_3 = 6$, $m_4 = 3$, $m_5 = 4$
  - Control word size = 5 + 4 + 6 + 3 + 4 = 22 bits.

$25 \le 2^5$ $15 \le 2^4$
$40 \le 2^6$ $5 \le 2^3$
$15 \le 2^4$

Control Unit Design for MIPS32

MIPS32 Instruction Encoding



rs, if present, always occupies bits 25..21

rt, if present, always occupies bits 20..16

rd, occupies bits 15..11

Imm16 occupies bits 15..0

Imm26 occupies bits 25..0

- The register operands as well as the 16-bit and 26-bit immediate data are retrieved and processed in case they are required later.

A Simple Implementation of MIPS32

- We consider the integer instructions and data path of MIPS32.
- Basic idea:
  - Different instructions require different number of register operands and immediate data (16 bits or 26 bits).
  - Relative positions of register encodings and immediate data are the same across instructions.

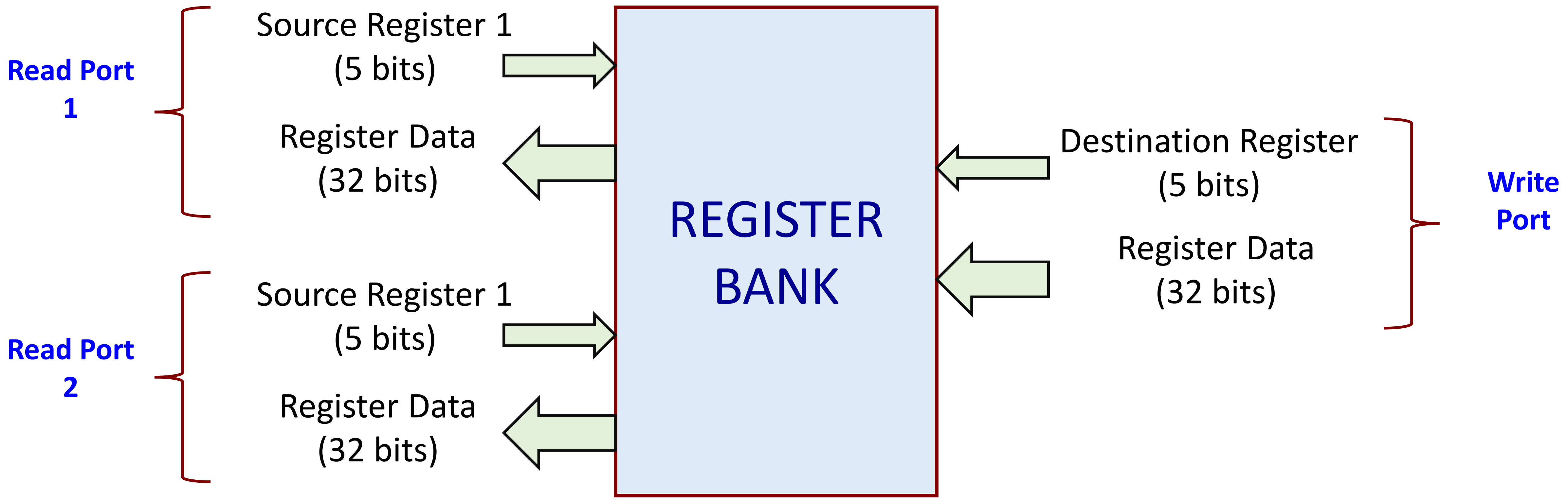
- **A Naïve Approach:**
  - After fetching and decoding an instruction, identify the exact register(s) and/or immediate operands to use, and handle them accordingly.
  - The number of register fetches and immediate operand processing will vary from instruction to instruction.
  - We do not utilize the possible overlapping of operations to make instruction execution faster.
    - Before instruction decoding is complete, fetch the register operands and immediate data in case they are required later.

An Assumption

- An instruction can have up to two source operands:

  ADD R1, R5, R10

  LW R5, 100(R6)
- There are 32 32-bit integer registers, *R0* to *R31*.
  - We design the register bank in such a way that two registers can be read simultaneously (i.e. there are 2 read ports).
  - We shall later see that performance can be improved by adding a write port (i.e. 2 reads and 1 write operations are possible per cycle).



Two register reads can be carried out simultaneously.

- **A Speculative Approach:**
  - We try to eliminate the time required to fetch the register operands and process the immediate data.
  - When an instruction is decoded, at the same time we fetch the register operands and also process the immediate data (i.e. sign extend).
    - Possible because their locations in the instruction word are fixed.
    - If the operands are required, they are already available (no extra time required).
    - If the operands are not required, they are ignored.

MIPS32 Instruction Cycle

- We divide the instruction execution cycle into five steps:

a)	IF	: Instruction Fetch
b)	ID	: Instruction Decode / Register Fetch
c)	EX	: Execution / Effective Address Calculation
d)	MEM	: Memory Access / Branch Completion
e)	WB	: Register Write-back

- We now show the generic micro-instructions carried out in the various steps.

(a) IF : Instruction Fetch

- Here the instruction pointed to by *PC* is fetched from memory, and also the next value of *PC* is computed.
  - Every MIPS32 instruction is of 32 bits (i.e. 4 bytes).
  - For a branch instruction, new value of the *PC* may be the target address. So *PC* is not updated in this stage; new value is stored in a register *NPC*.

IF:

```
IR    ←  Mem [PC];
NPC  ←  PC + 4;
```

(b) ID : Instruction Decode

- The instruction already fetched in *IR* is decoded.
    - *Opcode* is 6-bits: bits 31..26, with optional *function specifier*: bits 5..0
    - First source operand *rs*: bits 25..21, second source operand *rt*: bits 20..16
    - 16-bit immediate data: bits 15..0
    - 26-bit immediate data: bits 25..0
- Decoding is done in parallel with reading the register operands *rs* and *rt*.
    - Possible because these fields are in a fixed location in the instruction format.
- In a similar way, the immediate data can be sign-extended.

ID:

```
A     ← Reg [rs];
B     ← Reg [rt];
Imm   ← (IR_15)^16 ## IR_15..0             // sign extend 16-bit immediate field
Imm1  ← (IR_26)^6 ##IR_25..0 ## 00         // pad 2 0's to 26-bit immediate field
```

$A \leftarrow Reg[rs];$

$B \leftarrow Reg[rt];$

$Imm \leftarrow (IR_{15})^{16}\ \#\#\ IR_{15..0}$ // sign extend 16-bit immediate field

$Imm1 \leftarrow (IR_{26})^{6}\ \#\#IR_{25..0}\ \#\#\ 00$ // pad 2 0's to 26-bit immediate field

A, B, Imm, Imm1 are temporary registers.

(c) EX: Execution / Effective Address Computation

- In this step, the ALU is used to perform some calculation.
  - The exact operation depends on the instruction that is already decoded.
  - The ALU operates on operands that have been already made ready in the previous cycle.
- We show the micro-instructions corresponding to the type of instruction.

Memory Reference:

ALUOut ← A + Imm;

Example: LW R3, 100(R8)

Register-Register ALU Instruction:

ALUOut ← A func B;

Example: SUB R2, R5, R12
[operation specified by func field (bits 5..0)]

Register-Immediate ALU Instruction:

ALUOut ← A func Imm;

Example: SUBI R2, R5, 524
[operation specified by func field (bits 5..0)]

Branch:

ALUOut ← NPC + Imm1;

cond ← (A op 0);

Example: BEQZ R2, Label
[op is ==]

(d) MEM: Memory Access / Branch Completion

- The only instructions that make use of this step are loads, stores, and branches.
    - The load and store instructions access the memory.
    - The branch instruction updates *PC* depending upon the outcome of the branch condition.

Load instruction:

```
PC   ← NPC;
LMD  ← Mem [ALUOut];
```

Store instruction:

```
PC   ← NPC;
Mem [ALUOut]  ←  B;
```

Branch instruction:

```
if (cond) PC ← ALUOut;
else  PC ← NPC;
```

Other instructions:

```
PC   ← NPC;
```

(e) WB: Register Write Back

- In this step, the result is written back into the register file.
  - Result may come from the ALU.
  - Result may come from the memory system (viz. a LOAD instruction).
- The position of the destination register in the instruction word depends on the instruction → *already known after decoding has been done*.

	31 26	25 21	20 16	15 11	10 6	5 0
R-type	opcode	rs	rt	rd	shamt	funct

	31 26	25 21	20 16	15 0
I-type	opcode	rs	rt	Immediate Data

Register-Register ALU Instruction:

Reg [rd] ← ALUOut;

Register-Immediate ALU Instruction:

Reg [rt] ← ALUOut;

Load Instruction:

Reg [rt] ← LMD;

Some Example Instruction execution

ADD R2, R5, R10

IF	IR ← Mem [PC]; NPC ← PC + 4;
ID	A ← Reg [rs]; B ← Reg [rt];
EX	ALUOut ← A + B;
MEM	PC ← NPC;
WB	Reg [rd] ← ALUOut;

ADDI R2, R5, 150

IF	IR ← Mem [PC]; NPC ← PC + 4;
ID	A ← Reg [rs]; Imm ← $(IR_{15})^{16}$ ## $IR_{15..0}$
EX	ALUOut ← A + Imm;
MEM	PC ← NPC;
WB	Reg [rt] ← ALUOut;

LW R2, 200 (R6)

IF	IR ← Mem [PC];
	NPC ← PC + 4;

ID	A ← Reg [rs];
	Imm ← $(IR_{15})^{16}$ ## $IR_{15..0}$

EX	ALUOut ← A + Imm;

MEM	PC ← NPC;
	LMD ← Mem [ALUOut];

WB	Reg [rt] ← LMD;

SW R3, 25 (R10)

IF	IR ← Mem [PC];
	NPC ← PC + 4;

ID	A ← Reg [rs];
	B ← Reg [rt];
	Imm ← $(IR_{15})^{16}$ ## $IR_{15..0}$

EX	ALUOut ← A + Imm;

MEM	PC ← NPC;
	Mem [ALUOut] ← B;

WB	-

BEQZ R3, Label

IF	IR ← Mem [PC];
	NPC ← PC + 4;

ID	A ← Reg [rs];
	Imm ← $(IR_{15})^{16}$ ## $IR_{15..0}$

EX	ALUOut ← NPC + Imm1;
	cond ← (A == 0);

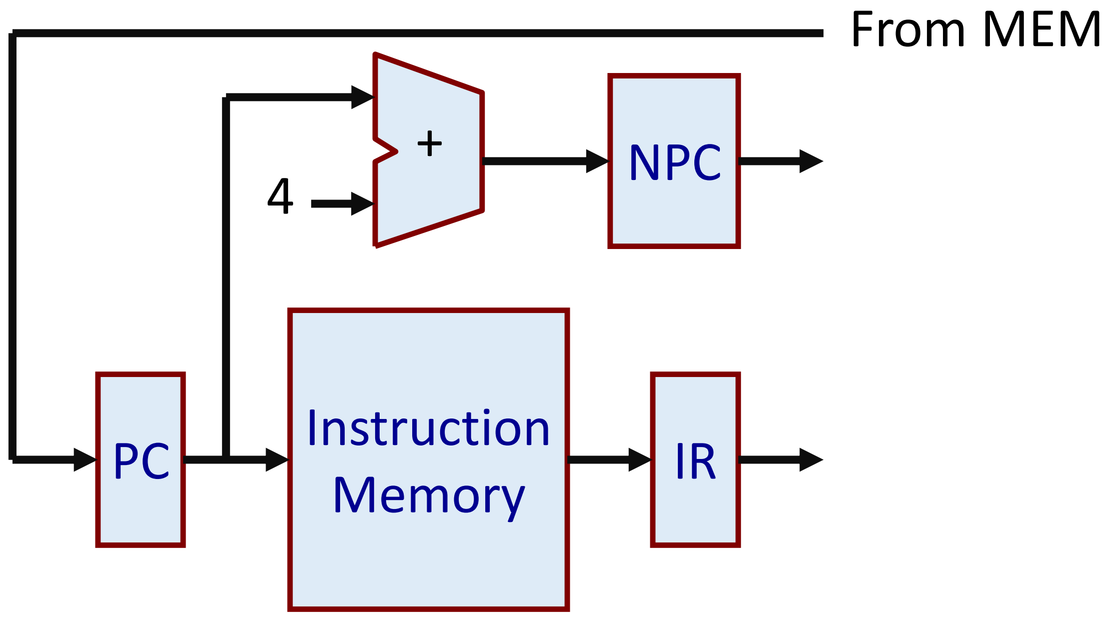
MEM	PC ← NPC;
	if (cond) PC ← ALUOut;

WB	-

Data Path Design of MIPS32

- We now show the data path for the five steps as mentioned for executing MIPS32 instructions.
  - *Assume that there is no pipelining*.
  - Also known as single-cycle implementation --- only after one instruction is finished can the next instruction start.
- Later on we shall extend the data path for pipelined implementation.
  - We shall discuss various pipelining related issues and techniques for faster execution of instructions.

The IF Stage

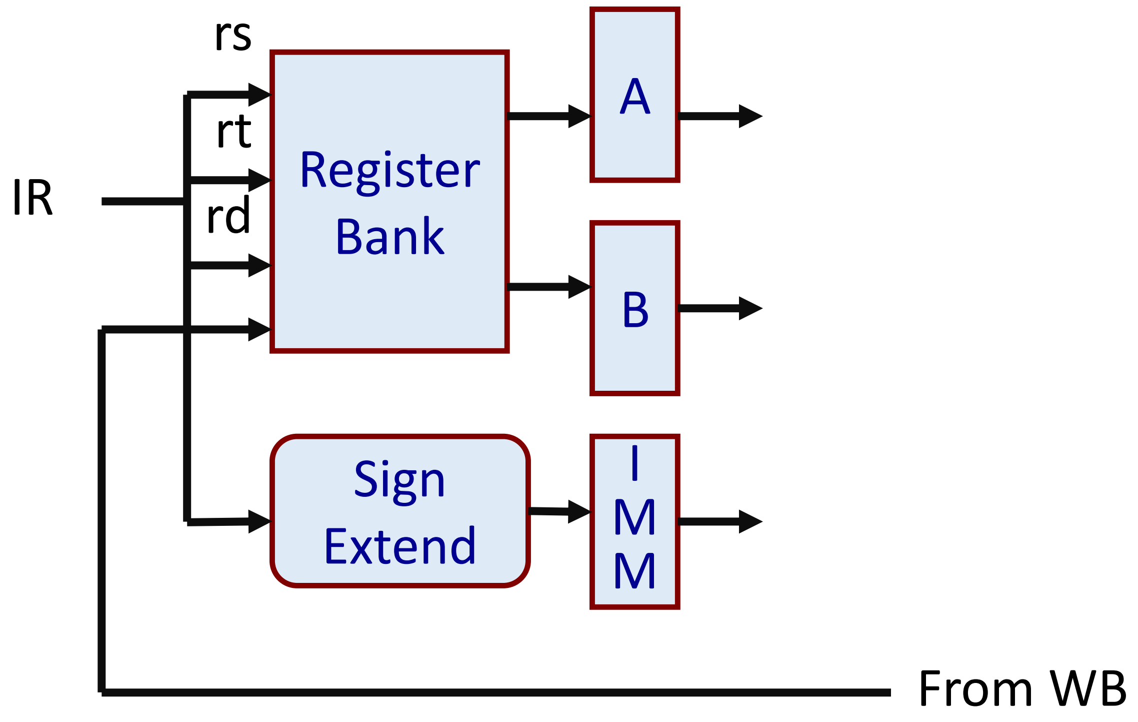


IR ← Mem [PC];

NPC ← PC + 4;

- 32-bit PC
- 32-bit NPC
- 32-bit IR
- 32-bit adder

The ID Stage



A $\leftarrow$ Reg [rs];

B $\leftarrow$ Reg [rt];

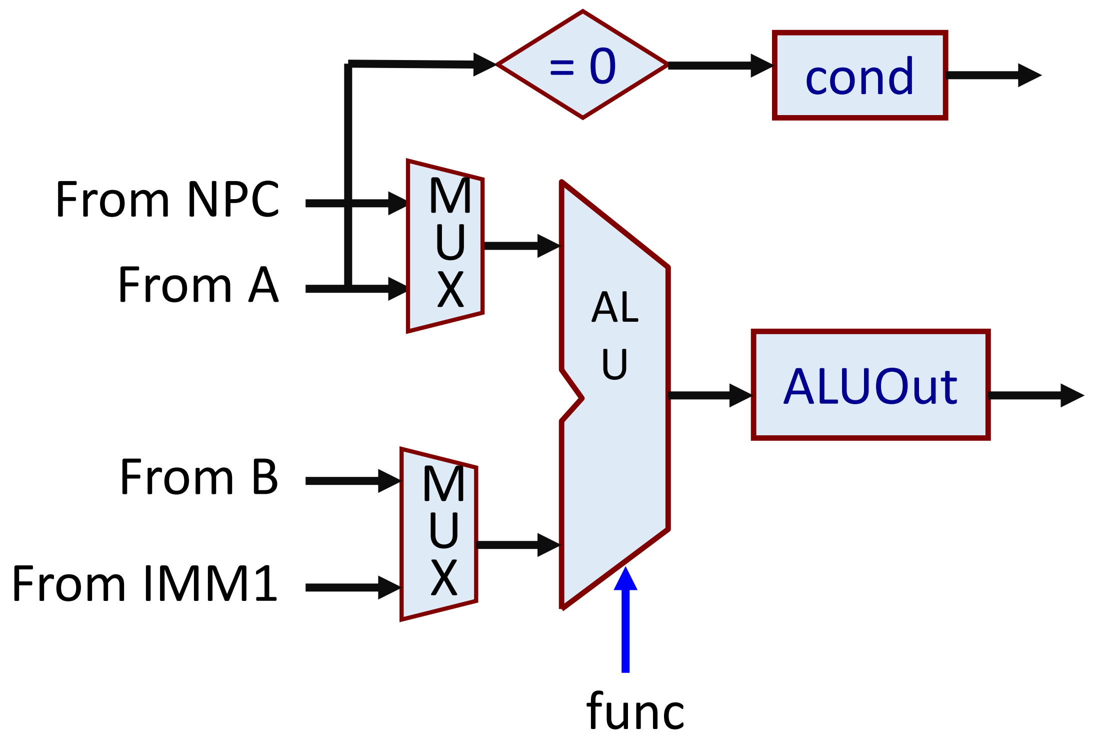
Imm $\leftarrow$ $(IR_{15})^{16}$ ## $IR_{15..0}$

Imm1 $\leftarrow$ $(IR_{25})^{6}$ ## $IR_{25..0}$ ## 00

The IMM1 calculation is not shown.

- 32-bit A
- 32-bit B
- 32-bit IMM

The EX Stage



- 32-bit ALUOut
- 1-bit cond
- 32-bit 2x1 MUX

Memory Reference:

ALUOut ← A + Imm;

Register-Register ALU Instruction:

ALUOut ← A func B;

Register-Immediate ALU Instruction:

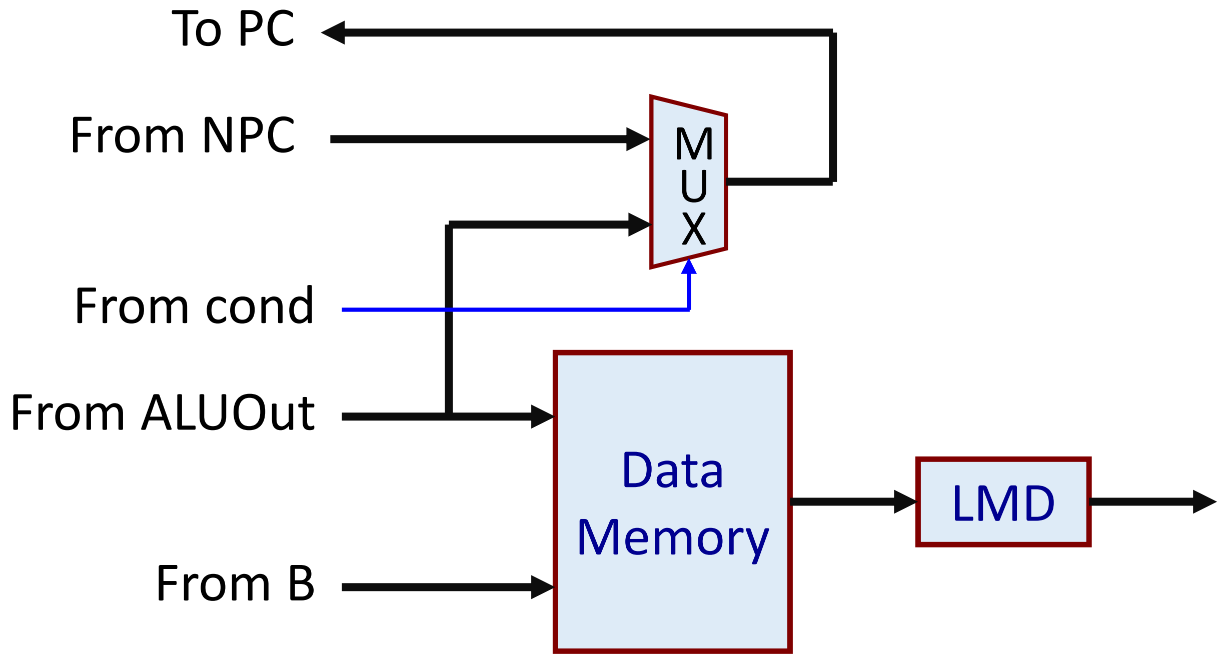
ALUOut ← A func Imm;

Branch:

ALUOut ← NPC + Imm1;

cond ← (A op 0);

The MEM Stage



- 32-bit LMD
- 32-bit MUX

Load instruction:

PC ← NPC;

LMD ← Mem [ALUOut];

Store instruction:

PC ← NPC;

Mem [ALUOut] ← B;

Branch instruction:

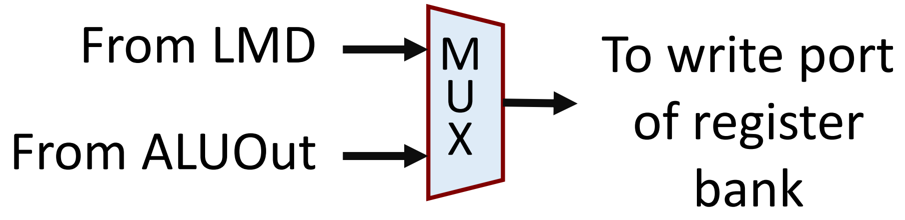
if (cond) PC ← ALUOut;

else PC ← NPC;

Other instructions:

PC ← NPC;

The WB Stage



- 32-bit MUX

Register-Register ALU Instruction:

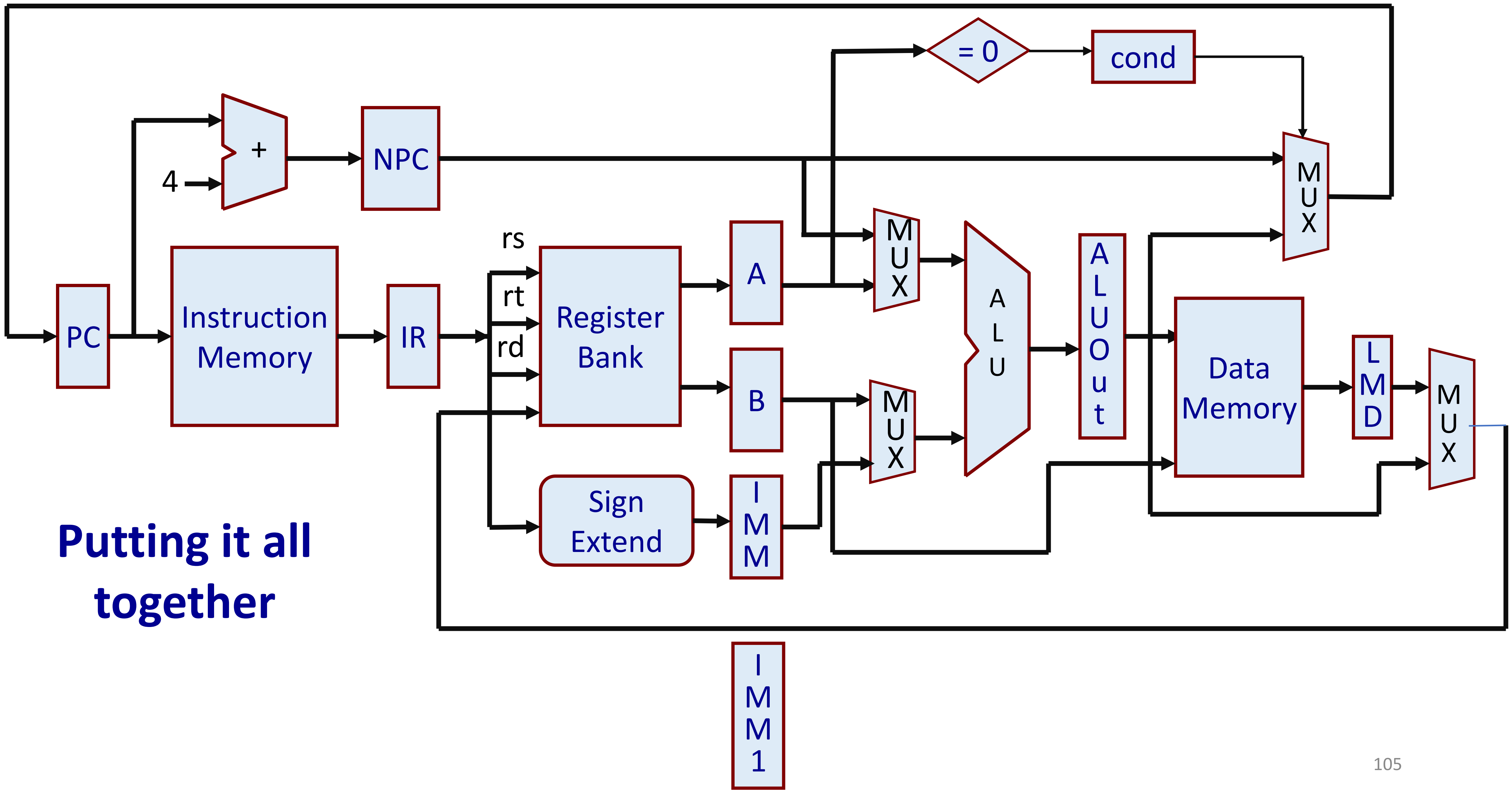
Reg [rd] ← ALUOut;

Register-Immediate ALU Instruction:

Reg [rt] ← ALUOut;

Load Instruction:

Reg [rt] ← LMD;



Putting it all together

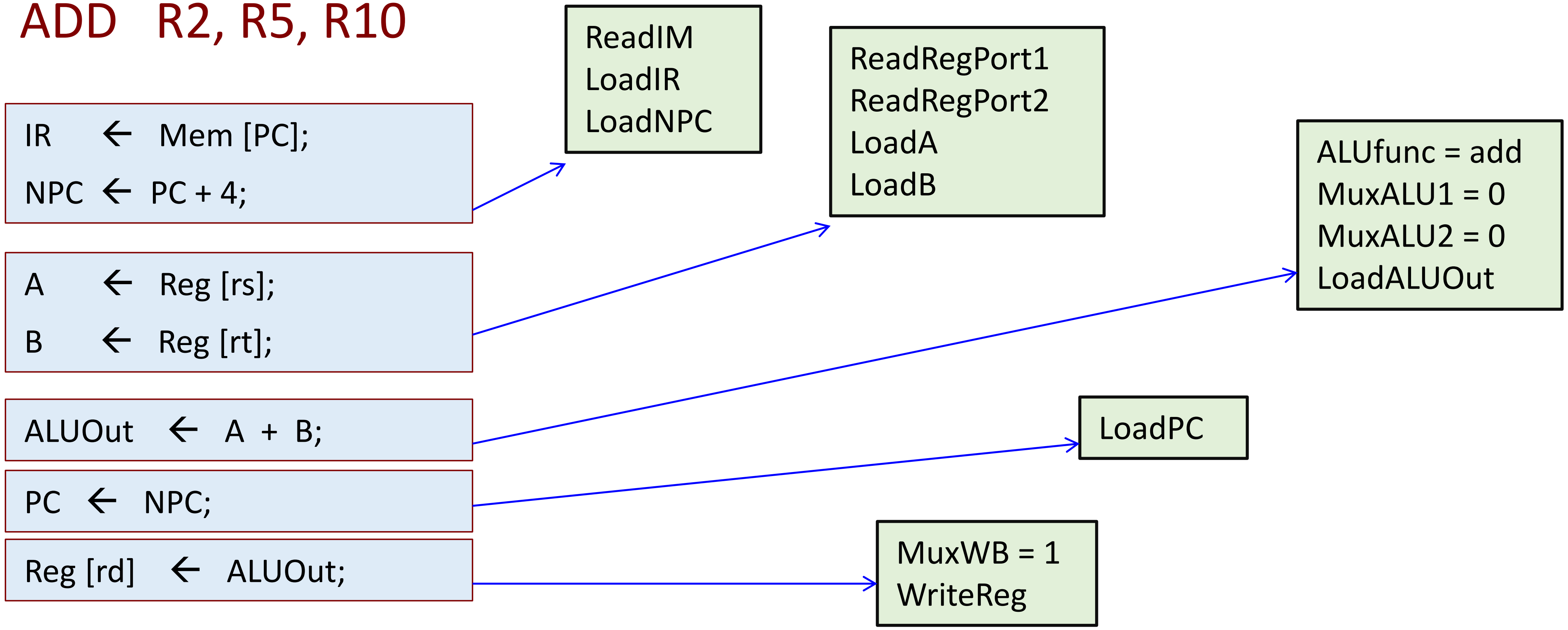
Simplicity of the Control Unit Design

- Due to the regularity in instruction encoding and simplicity of the instruction set, the design of the control unit becomes very easy.
- Control signals in the data path:

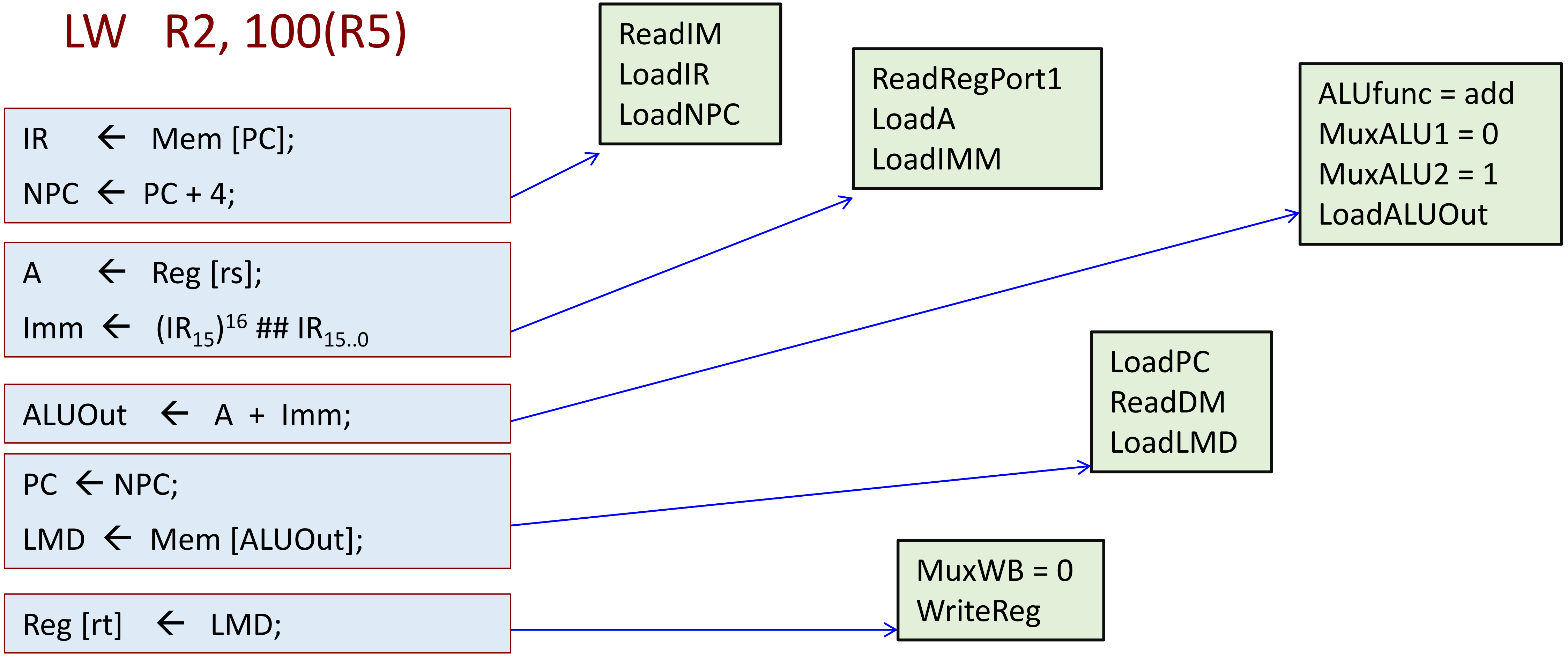
a) LoadPC
b) LoadNPC
c) ReadIM
d) LoadIR
e) ReadRegPort1
f) ReadRegPort2
g) LoadA
h) LoadB
i) LoadIMM
j) MuxALU1
k) MuxALU2
l) ALUfunc
m) LoadALUOut
n) MuxPC
o) ReadDM
p) WriteDM
q) LoadLMD
r) MuxWB
s) WriteReg

Control Signals for Some Instructions

ADD R2, R5, R10



LW R2, 100(R5)



IR ← Mem [PC];
NPC ← PC + 4;
	ReadIM
	LoadIR
	LoadNPC

A ← Reg [rs];
Imm ← $(IR_{15})^{16}$ ## $IR_{15..0}$
	ReadRegPort1
	LoadA
	LoadIMM

ALUOut ← A + Imm;	ALUfunc = add
	MuxALU1 = 0
	MuxALU2 = 1
	LoadALUOut

PC ← NPC;
LMD ← Mem [ALUOut];
	LoadPC
	ReadDM
	LoadLMD

Reg [rt] ← LMD;	MuxWB = 0 WriteReg