alex-levesque.com

ELEC 274 — Computer Architecture

Winter 2026

Based on lectures by N. Manjikian – Queen's University

Notes written by Alex Lévesque

These notes are my own interpretations of the course material and they are not endorsed by the lecturers.
Feel free to reach out if you point out any errors.

Contents

1 Preface

Grading Scheme:

Textbook:

Comments:

-

2 Assembly Guide

In Nios II, every instruction is 32 bits wide. Since memory addresses are assigned to individual bytes (8 bits), we know that each instruction occupies a 4-byte block. Therefore, in a routine, the starting address of the next instruction must be exactly 4 bytes higher than the previous one.

In register transfer notation, use [...] to denote contents of a location. Use $\leftarrow$ to denote transfer to a destination. Example: $R2 \leftarrow [LOC]$. We can also do $R4 \leftarrow [R2] + [R3]$

2.1 Data Movement & Memory Access

These instructions are used to move data between registers or between registers and memory

`mov` (move register): `mov dest., src.`

- Definition: copies the contents of one register into another register
- Purpose: Used when you need to duplicate a value that is currently in a register to use it elsewhere without modifying the original

`movi` (move immediate): `mov dest., IMM16`

- Definition: Loads a 16-bit constant (immediate) value into a register. The value is sign-extended to 32 bits.
- Purpose: Used to initialize a register with a small number (like 0, 1, or small offsets)

`movia` (move immediate address): `mov dest., label/32-bit address`

- Definition: A macro (pseudo-instruction) that loads a full 32-bit value (typically a memory address) into a register
- Purpose: Since standard instructions can only handle 16-bit numbers at a time, movia is essential for loading the addresses of labels so the program knows where data is located in memory

`ldw` (load word): `ldw dest., byte_offset(base address)`

- Definition: Reads a 32-bit word from a specific memory address and loads it into a destination register
- Purpose: Used to retrieve data stored in RAM so the processor can operate on it

`stw` (store word): `stw source, byte_offset(base address`

- Definition: Writes the 32-bit value currently in a register to a specific memory address
- Purpose: Used to save the results of calculations back into RAM for later use

Byte offsets are used to provide a flexible and efficient way to access specific memory locations relative to a known starting point

2.2 Arithmetic Operations

These instructions perform mathematical calculations.

`add` (add): `add dest., src., src.`

- Definition: adds the contents of two source registers together and stores the result in a destination register
- Purpose: Used for standard addition of variables

`addi` (add immediate): `addi dest., src., IMM16`

- Definition: adds the value of a source and a 16-bit constant, storing the result in the destination
- Purpose: commonly used to increment counters or move points to the next item in a list

`subi` (subtract immediate): `subi dest., src., IMM16`

- Definition: Subtracts a 16-bit constant value from source and stores the result in the destination
- Purpose: used to decrement counters or adjust values downwards by a fixed amount

2.3 Control Flow (Branching)

These instructions change the order in which the code executes (loops and if-statements). Branch instruction causes repetition of body. Many branches are conditional, as seen below.

`br` (branch): `br LABEL`

- Definition: unconditionally jumps to the instruction located at LABEL
- Purpose: used to force a loop to repeat or to skip over a section of code entirely

`beq` (branch if equal): `beq src., src., LABEL`

- Definition: compares registers; if they are equal, the program jumps to LABEL
- Purpose: often used to check loop termination conditions

`bne` (branch if not equal): `bne src., src., LABEL`

`bge`, `ble`, `bgt`, `blt` (branch if $\geq, \leq, >, <$): `opcode src., src., LABEL`

- Definition: compares registers; if they are $\geq, \leq, >, <$, the program jumps to LABEL
- Purpose: often used to check loop termination conditions

2.4 Subroutines (Functions)

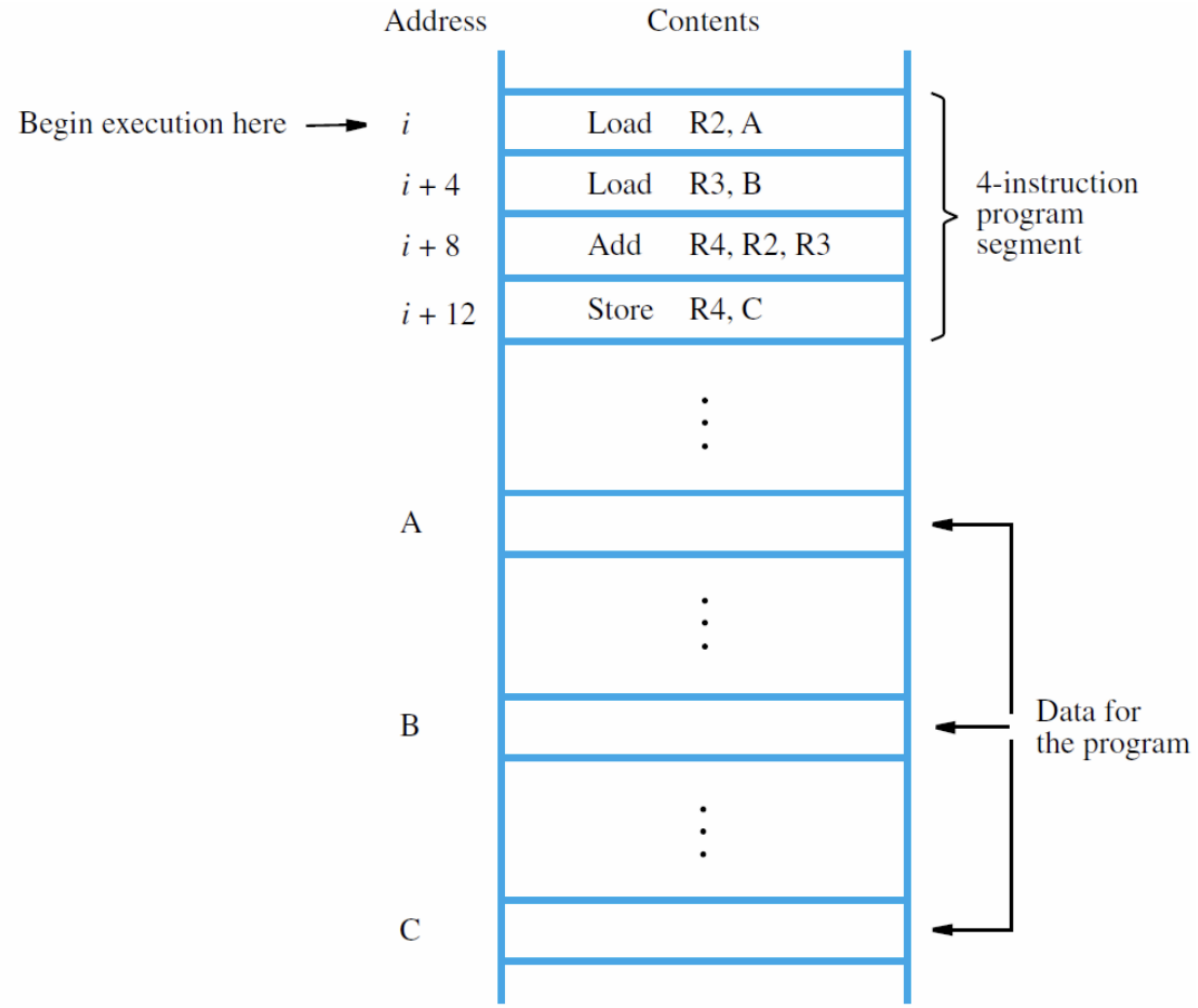
These instructions are used to call and return from functions

`call` (call subroutine): `call LABEL`

- Definition: jumps to LABEL and automatically saves the address of the next instruction into the return address register r31
- Purpose: allows the program to execute a separate block of code and remember where to come back to when it's done

`ret` (return):

- Definition: jumps to the address stored in the return address register
- Purpose: used at the end of function to send the processor back to the point in the main code immediately following the call instructions

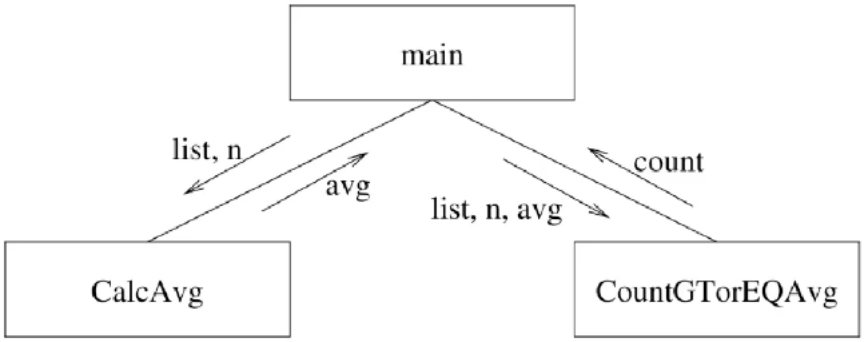


2.5 Preparation of a Subroutine

Process:

- analyze the problem to understand overall computing task
- partition the problem into smaller, more manageable components or subtasks
- characterize the nature of data provided as input and generated as output
- select appropriate data structures to be used during the execution of the program
- define and organize software modules that encompass the subtasks from the partitioning step, and specify the parameter passing between those modules
- refine the details of the various software modules in pseudocode form
- translate the pseudocode description into the target language for implementation
- generate executable code and test its operations for correctness

Example: Calculate average of list of numbers, then determine number of values in the list are greater than average, and less than average. This example focuses on the subroutine CountGTorEQAvg



Pseudocode:

```
CountGTorEQAvg(list, n, avg):: // r2, r3, r4
    count = 0 // r6
    for i = 0 to n-1 do
        if (list[i] >= avg) then
            count = count + 1
        end if
    end for
    return count // r2
```

Code:

```
CountGTorEQAvg:
    // allocate space on stack for local variables
    subi   sp, sp, 12
    stw    r3, 8(sp) // save n
    stw    r5, 4(sp) // save list element
    stw    r6, 0(sp) // save count

    // initialize count = 0
    movi   r6, 0

count_loop:
// load current list elements: r2 points to list
// compare current element r3 with avg r4, skip increment if r3<r4
if:
    ldw    r3, 0(r2)
    blt    r5, r4, end_if

// if list[i]>=avg, increment count
then:
    addi   r6, r6, 1

end_if:
// move to next element in the list (4 byte increment)
    addi   r2, r2, 4
// decrement loop counter n
    subi   r3, r3, 1
```

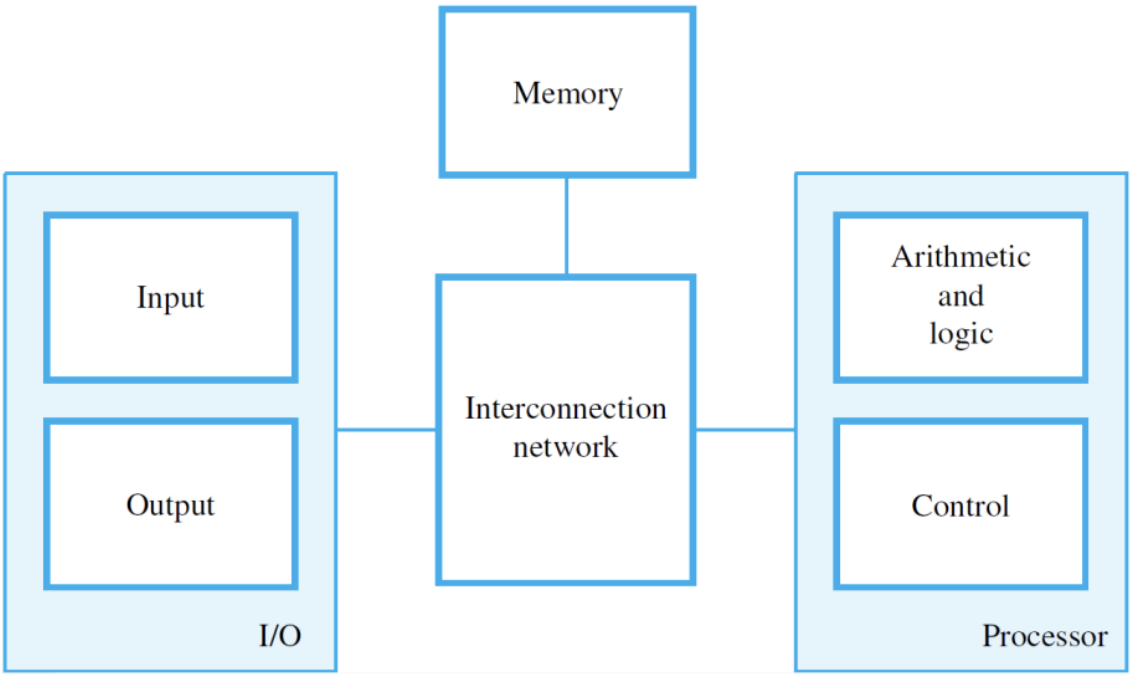
```
// if counter > 0, continue loop
    bgt    r3, r0, count_loop

// store result in r2 (return value)
    mov    r2, r6

// restore saved registers
    ldw    r3, 8(sp)
    ldw    r5, 4(sp)
    ldw    r6, 0(sp)
    addi   sp, sp, 12

    ret
```

3 Basic Structure of Computers

3.1 Computer Architecture

Definition: Computer architecture is the specification of a set of instructions and behaviour of hardware units

Functional Units: Computers consist of 5 basic units: input, memory, arithmetic and logic, output, and control. The interconnection network supports transfer of information between units. The **processor** includes arithmetic and logic with control. The **I/O System** includes input and output units together.



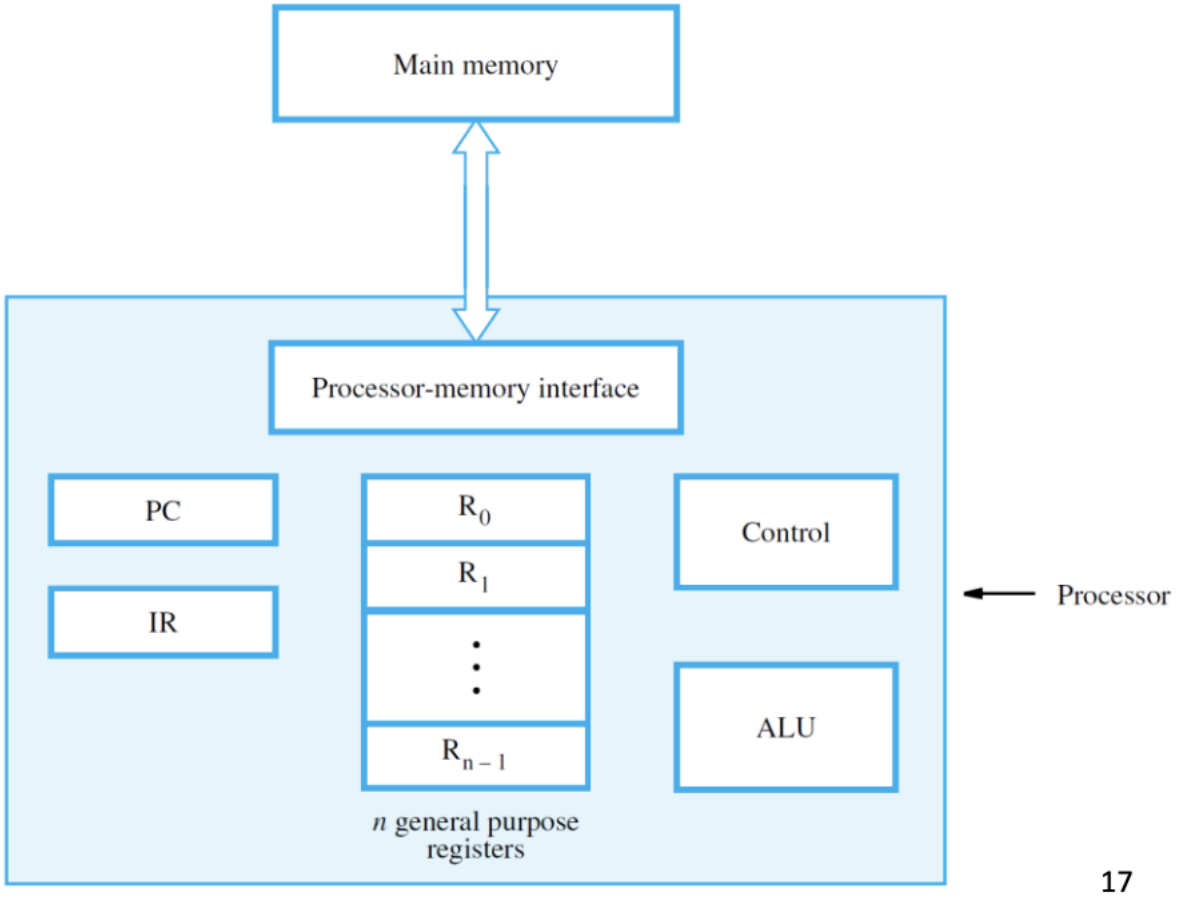
Input: Accepts coded information (in binary) for processing. Ex: mouse, keyboard, etc.

Memory: Stores programs (instruction lists) and data. Ex: RAM, SSD/HDD, cache

ALU: Performs arithmetic $(+, -, \cdot, /)$ and logic $(\wedge, \vee, \neg)$. Ex: high-speed registers for holding operands

Output: Presents processed results. Ex: displays, printers, and audio devices

Control: Coordinates all other units by sending timing and state signals. Ex: control circuits and control lines



How They Interact

1. **Instruction Fetching:** The **Control Unit** uses the **Program Counter (PC)** to find the address of the next instruction in **Memory**. This instruction is moved into the **Instruction Register (IR)** within the processor. During execution of each instruction, PC register is incremented by 4.
2. **Decoding:** The Control Unit interprets (decodes) the instruction in the IR to determine what action is required.
3. **Data Transfer:** Depending on the instruction, data may be moved from an **Input Unit** to Memory, or from Memory to **Processor Registers** (Load instruction).
4. **Processing:** If the instruction involves math or logic, the Control Unit directs the **ALU** to perform the operation using operands stored in registers.
5. **Storing and Outputting:** The result of a calculation is either kept in a register or written back to **Memory** (Store instruction). Finally, the **Output Unit** may transfer these results to a user or external device.

Instructions Running Cycle:

1. Fetch: The CPU fetches the instructions from memory. It uses the PC to know the memory address of the next instruction which is then loaded into the IR
2. Execute: The CPU carries out what the instruction says, could be ALU, jump/branch, moving data, etc.

3.2 Memory Organization

Hierarchy and Storage Types

Memory is organized into different levels to balance speed, capacity, and cost.

Primary Memory: This is fast, electronic memory composed of semiconductor storage cells. It is essential for storing programs and data currently in use.

Cache Memory: A smaller, faster electronic memory located on the same chip as the processor. It holds copies of instructions and data from the main memory that were recently used or are likely to be used soon, significantly speeding up access.

Secondary Storage: This provides large-capacity storage that retains information even when the power if off. While it is less expensive per bit, i tis generally slower than primary memory and has traditionally been based on magnetic or optical devices, though it now includes flash memory.

Physical and Logical Organization

The physical structure of memory dictates how the processor interacts with it.

- Binary Representation: Information is stored in bits
- Words: Bits are grouped into multi-bit "words" (typically 32 bits) to allow the processor to access multiple bits simultaneously for efficiency
- Addressing: Each word location has a unique address, numbered consecutively starting at 0
- Random Access Memory: Memory is organized so that any location can be accessed in a fixed, short amount of time, regardless of where the data is physically located

4 Instruction Set Architecture

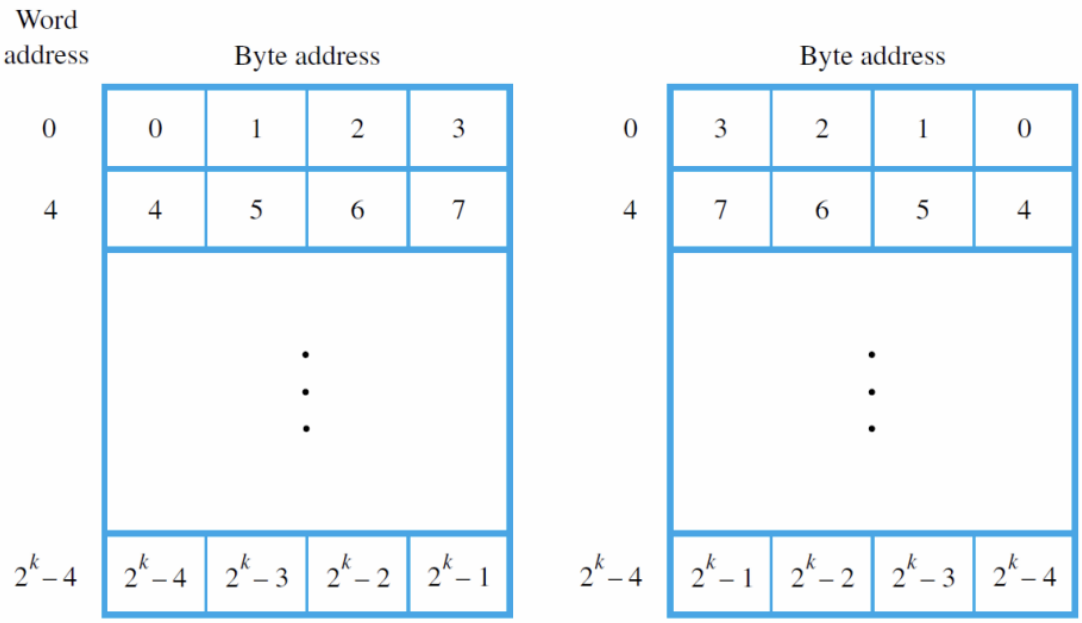
4.1 Memory Locations and Addresses, and Instruction Execution

The memory of a computer consists of many millions of storage cells, each of which can store a *bit* of information having the value of 0 or 1.

A common word length is 32 bits. Numbers 0 to $2^k - 1$ are used as addresses for successive locations in the memory. Byte size is always 8 bits, but word length may range from 16 to 64 bits.

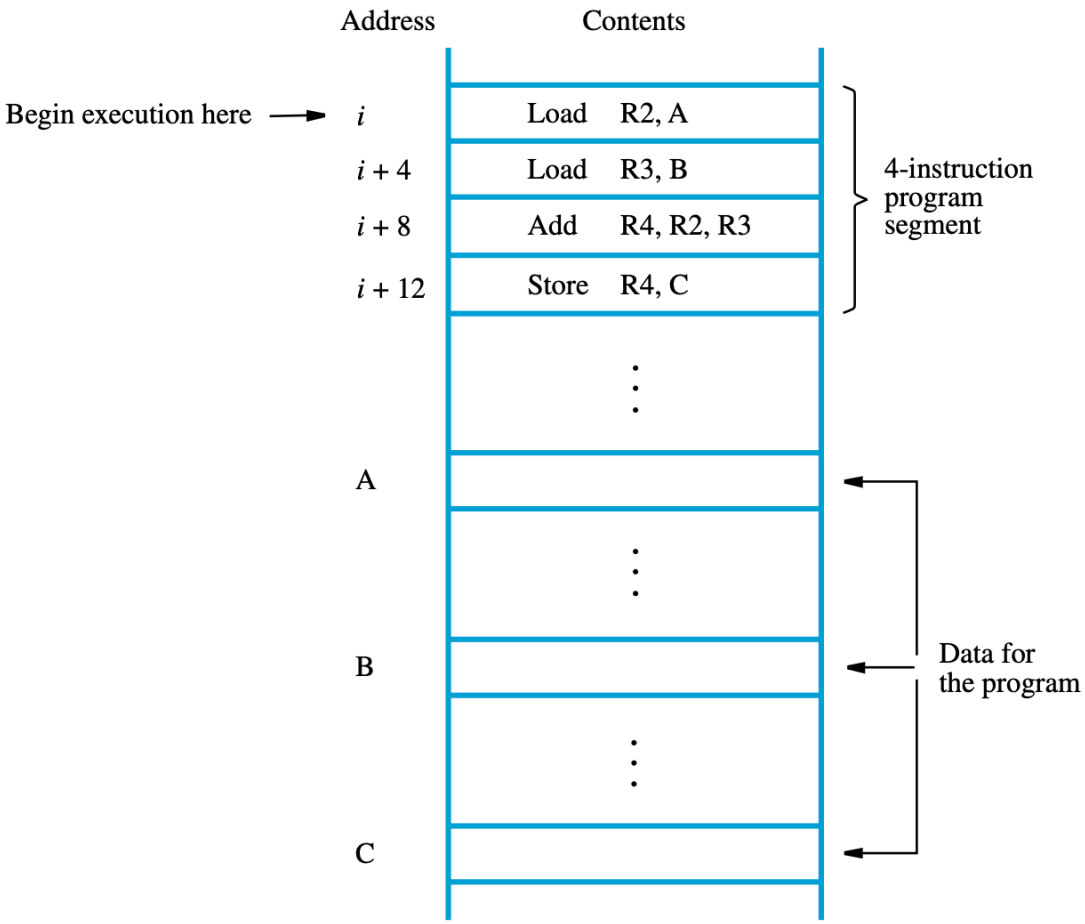
Byte locations have addresses $0, 1, 2, \ldots$, and word locations have addresses $0, 4, 8, \ldots$. We provide a byte-addressable memory that assigns an address to each byte. We have two ways to assign byte address across words.

There are two ways that byte addresses can be assigned across words. **Big-endian:** is used when lower byte addresses are used for the more significant bytes. **Little-endian** uses opposite order.



(a) Big-endian assignment (b) Little-endian assignment

For example, there are 4 instructions to execute `C = A + B`, and each instruction are in successive word locations. Since each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses $i + 4, i + 8$, and $i + 12$.

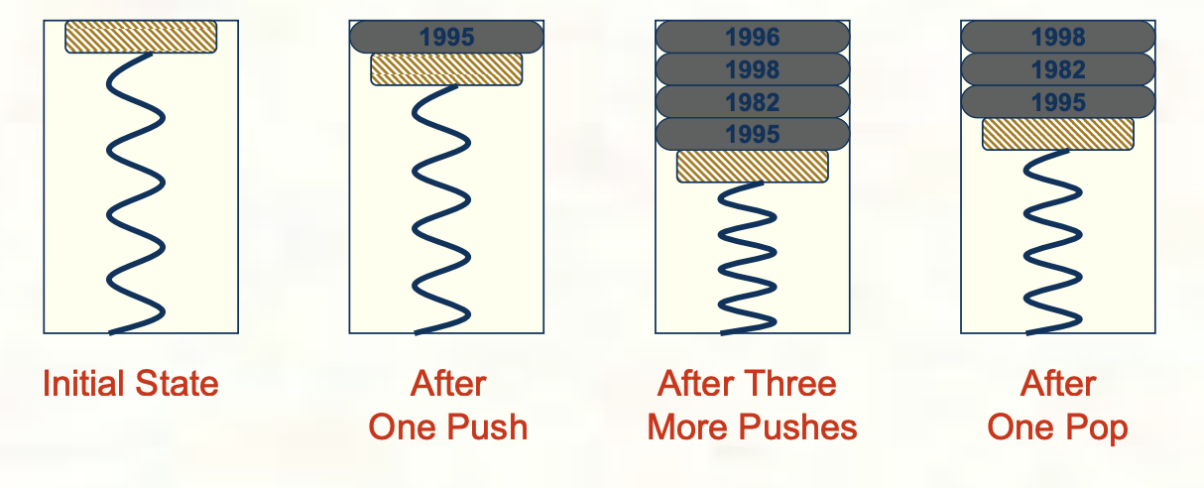


The processor contains a register PC which holds the address of the next instruction to be executed. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing*. Then a two phase execution procedure begins:

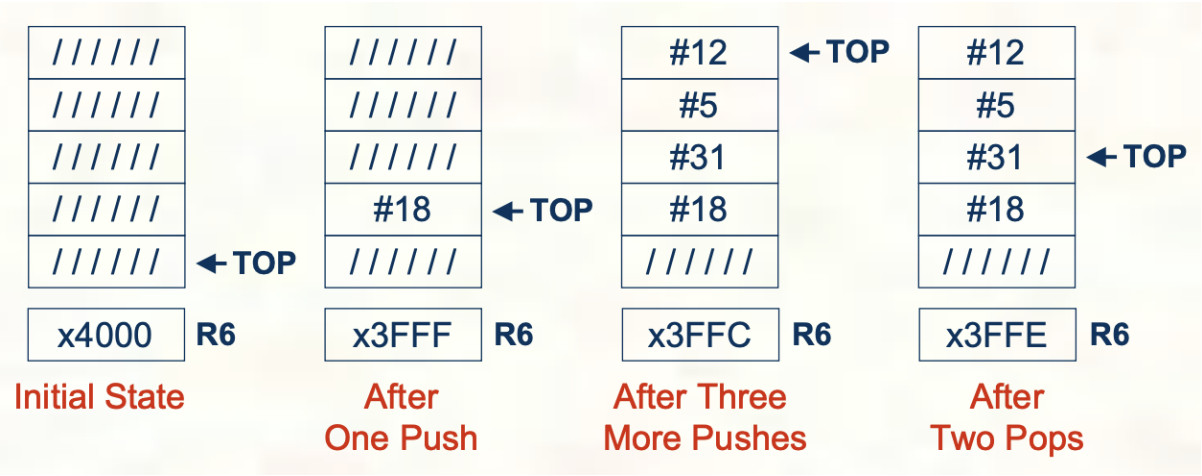
- instruction fetch

4.2 Stack and Subroutines

Definition: A **stack** is a LIFO list of data elements. The processor stack has a stack pointer `sp` register that points to top of the processor stack



The data in memory does not physically move when we push or pop a stack, only the stack point changes.



Definition: We use **subroutines** to avoid repeating a particular task's function.

When a subroutine is called, the processor must remember where to return after the subroutine finishes. This is handled by the stack.

4.3 Subroutine Linkage and Nesting

The way in which a computer makes it possible to call and return from subroutines is referred to as its *subroutine linkage* method.

We can simply save the return address in a specific location called the *link register*. When a subroutine completes the task, the Return instruction returns to the calling program by branching indirectly through the link register.

The Call instruction is just a special branch instruction that performs the following operations:

- Store the contents of the PC in the link register
- Branch to the target address specified by the Call instruction

The Return instruction is a special branch instruction that performs: Branch to the address contained in the link register.

Subroutine Nesting

When two subroutines are nested, the return address of the second must not overwrite the return address of the first one. We can stack these return addresses LIFO onto the processor stack to avoid overwriting.

5 Basic Input/Output

An I/O device interface is a circuit situated between a device and the interconnection network. It facilitates data transfer and the exchange of status and control information. It is a collection of registers:

- DATA register: holds the data being transferred to or from the device
- STATUS register: contains flags indicating the state of the device
- CONTROL register: allows the processor to configure device behavior

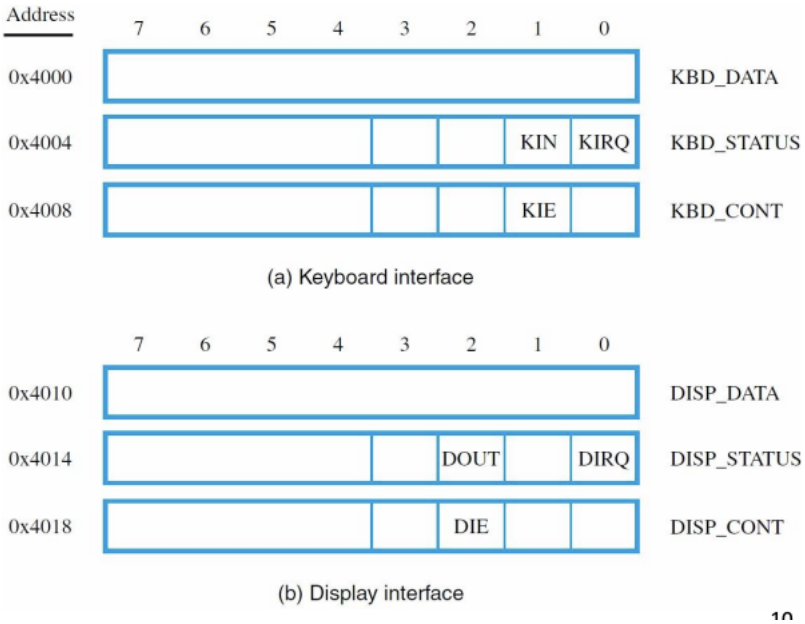
Assume:

1. Keyboard (Address 0x4000 - 0x4008):

- KBD DATA: Contains the ASCII character read.
- KBD STATUS: Contains the KIN flag (bit 1), which is set to 1 when a character is ready to be read. Reading KBD DATA automatically clears KIN.
- KBD CONT: Contains the KIE flag (bit 1) to enable keyboard interrupts.

2. Display (Address 0x4010 - 0x4018):

- DISP DATA: Destination for character to be displayed
- DISP STATUS: Contains the DOUT flag (bit 2), which is set to 1 when the display is ready to accept a new character. Writing to DISP DATA clears DOUT.
- DISP CONT: Contains the DIE flag (bit 2) to enable display interrupts.



5.1 Program-Controlled I/O (Polling)

Program-controlled I/O requires the processor to continuously check (poll) the status of a device to determine if it is ready for data transfer

Wait Loop: To read a character, the processor executes a "wait loop" reading the status register until the KIN flag is set.

```
// Initialization
```

```
Move R2, #LOC            ; R2 points to memory buffer
Move R3, #CR             ; Load Carriage Return ASCII

READ:
    // Poll Keyboard
    LoadByte R4, KBD_STATUS
    And R4, R4, #2           ; Check Bit 1 (KIN)
    Branch_if_Zero R4, READ  ; Loop if not ready

    LoadByte R5, KBD_DATA    ; Read Data (clears KIN)
    StoreByte R5, (R2)       ; Store in memory
    Add R2, R2, #1           ; Increment pointer

ECHO:
    // Poll Display
    LoadByte R4, DISP_STATUS
    And R4, R4, #4           ; Check Bit 2 (DOUT)
    Branch_if_Zero R4, ECHO  ; Loop if not ready

    StoreByte R5, DISP_DATA ; Write to Display (clears DOUT)

    // Check for Carriage Return to exit
    Branch_if_Equal R5, R3, EXIT
    Branch READ              ; Get next character
EXIT:
```

5.2 Interrupt-Based I/O

In polling, the processor is unable to perform computations while waiting for slow I/O devices. **Interrupts** solve this by allowing the I/O device to alert the processor via a hardware signal when it is ready. When interrupted, the processor can execute a specific Interrupt-Service Routine (ISR) when necessary

An ISR is different than a subroutine because it is not invoked by a Call instruction. It can occur at any time during execution (therefore asynchronous). The processor must save the PC and restore it exactly to instruction $i + 1$ after the ISR completes

Interrupts are controlled via flags in both the processor and device interfaces. At the processor level, The Processor Status (PS) register contains an IE (Interrupt Enable) bit. When an interrupt occurs, the processor automatically saves the PS to the IPS register and clears the IE bit to prevent immediate re-interruption. The return-from-interrupt instruction restores the state.

When handling multiple devices, we have:

- The ISR reads the status registers of all devices to see which IRQ bit is set
- The requesting device sends a code identifying itself, then the processor uses this code to look up the address of the specific ISR in an interrupt-vector table

The main setup must initialize pointers, enable interrupts at the device level, and finally enable interrupts at the processor level

```
START:
    Move R2, #LINE
    Store R2, PNTR          ; Initialize buffer pointer
    Clear R2
    Store R2, EOL           ; Clear end-of-line indicator

    // Enable Interrupts in Keyboard Interface
    Move R2, #2
    StoreByte R2, KBD_CONT

    // Enable Keyboard Interrupts in Processor
    MoveControl R2, IENABLE
    Or R2, R2, #2           ; Set specific bit for KBD
    MoveControl IENABLE, R2

    // Globally Enable Interrupts (IE bit in PS)
    MoveControl R2, PS
    Or R2, R2, #1
    MoveControl PS, R2

    // Continue with main computation...
```

The ISR must save the context (registers used) onto the stack, perform the I/O operation, check logic, restore context, and return

```
ILOC:
    Subtract SP, SP, #8     ; Make space on stack
    Store R2, 4(SP)         ; Save R2
    Store R3, (SP)          ; Save R3

    Load R2, PNTR           ; Load buffer pointer
    LoadByte R3, KBD_DATA   ; Read character (clears KIN)
    StoreByte R3, (R2)      ; Store in buffer
    Add R2, R2, #1          ; Increment pointer
    Store R2, PNTR          ; Update pointer variable

    // Check for Carriage Return (End of Line)
    Move R2, #CR
    Branch_if_Not_Equal R3, R2, RTRN

    // If CR, set EOL flag and disable keyboard interrupts
    Move R2, #1
    Store R2, EOL
    Clear R2
    StoreByte R2, KBD_CONT

RTRN:
    Load R3, (SP)           ; Restore R3
    Load R2, 4(SP)          ; Restore R2
    Add SP, SP, #8          ; Restore Stack Pointer
```

```
Return-from-interrupt
```

6 Software

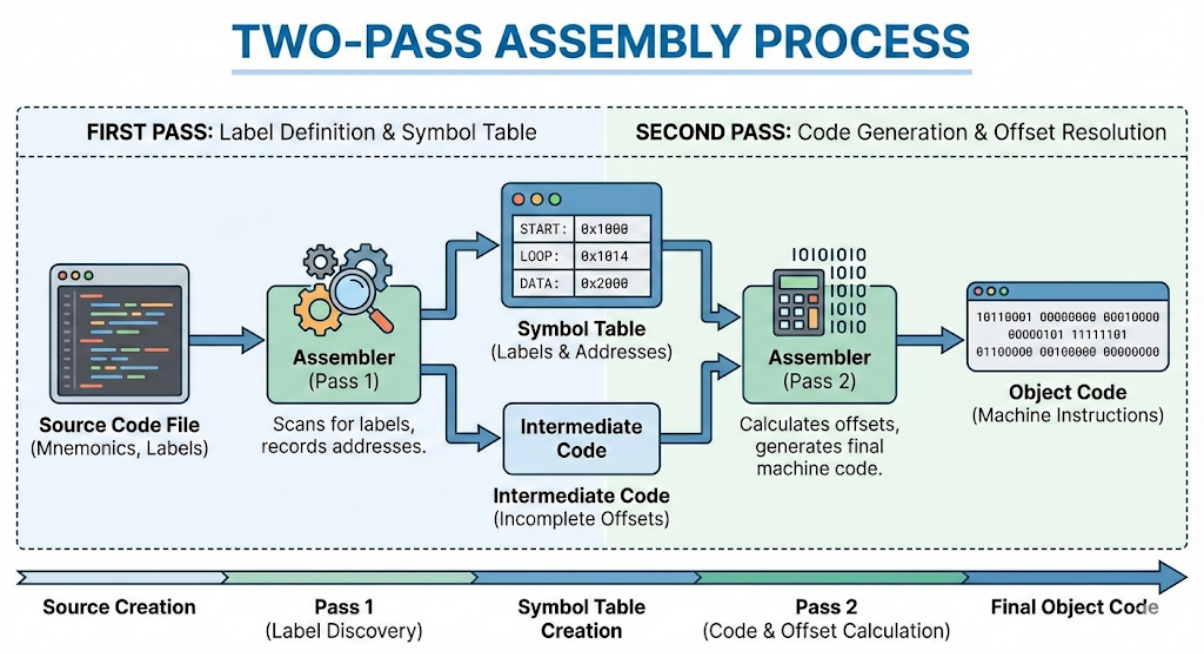
Outline:

- Assembler, compiler, and linker are tools for generating object programs
- Loader transfers programs into memory and handles execution initiation/termination
- Debugger identifies programming errors
- I/O and other tasks can be programming in multiple languages
- Operating system manages everything

6.1 Assembly Process

While the source program is created using a text editor, the **assembler** translates the source file to object code. It can recognize and interpret mnemonics, addressing modes for operands, and directives to define constants and allocate space in memory for data.

Sometimes an offset in memory cannot be found without knowing the target address. A **two-pass assembler** is employed to firstly generate all machine instructions and to enter labels and addresses into the symbol table, then since sizes of instructions are known, the second pass can calculate the unknown branch offsets.



Loader: The loader is invoked when a user types a command or clicks on an icon in a graphical user interface. This user input identifies the specific object file on the disk, which contains information on the starting location in memory and the total length of the program. The loader transfers the object program from the disk to the memory and branches to the starting address. Upon program termination, the loader recovers the space in memory and awaits the next command.

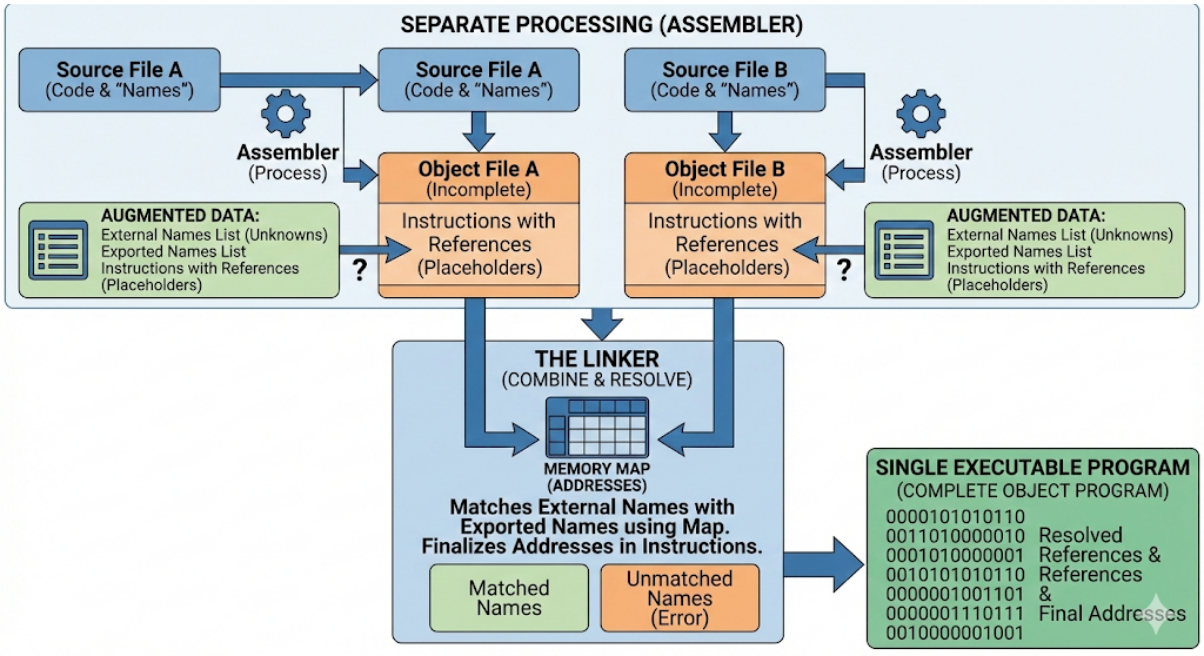
6.2 Separate Processing

It is important to distinguish between separate and parallel processing. While parallel processing divides the work of the hardware (handling computation and I/O at the

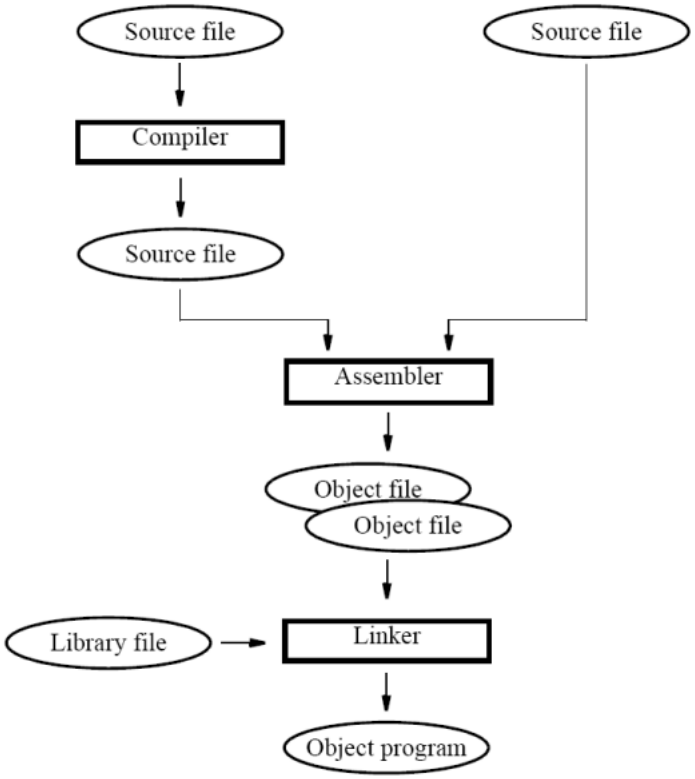
same time), **separate processing** divides the work of the programmer and the tools (assembling small files is faster than one giant one)

When two separate source files are passed through the assembler, it outputs an object file containing machine code, data, and section headers. If some instructions have references to subroutines in different source files, a **placeholder** is left there. The augmented data is like a cheat sheet that contains the offsets of each function with a placeholder.

The **Linker** then combines object files into an object program, constructs a map of the full program in memory, then determines the addresses of all names, allowing external names (unknowns) to be matched with exported names (placeholders). Any unresolved names are reported as errors.



Subroutines that are useful in one program are often useful in other programs. These subroutines can be linked for reuse or collected by the archiver into a library file (containing name information to aid in resolving references from calling program).


Source file
Source file
Compiler
Source file
Assembler
Object file
Object file
Library file
Linker
Object program

7 Basic Processor Unit

7.1 5-Step Execution Sequence

All RISC-style instructions in Chapter 5 follow a universal 5-step cycle. We must write these using Register Transfer Notation (RTN).

Fetch: $IR \leftarrow [[PC]]$, Read Memory, $IR \leftarrow$ Memory Data, $PC \leftarrow [PC] + 4$ (the processor fetches the instruction from memory and increments the counter to the next word)

Decode/Read: $RA \leftarrow [Rs]$; $RB \leftarrow [Rt]$

The execution steps (3, 4, and 5) vary by instruction type:

Instruction Type	**Step 3 (Execute)**	**Step 4 (Memory)**	**Step 5 (Write-back)**
ALU (Add, Sub, Mult, etc.)	$RZ \leftarrow [RA]$ op $[RB]$	$RY \leftarrow [RZ]$	$Rd \leftarrow [RY]$
Load $Rt, X(Rs)$	$RZ \leftarrow [RA] + X$	$RY \leftarrow [[RZ]]$	$Rt \leftarrow [RY]$
Store $Rt, X(Rs)$	$RZ \leftarrow [RA] + X; RM \leftarrow [RB]$	$[RZ] \leftarrow [RM]$	No action
Call *address*	$PC\_Temp \leftarrow [PC]; PC \leftarrow$ address	$RY \leftarrow [PC\_Temp]$	$LINK \leftarrow [RY]$
Return	$PC \leftarrow [RA]$	No action	No action

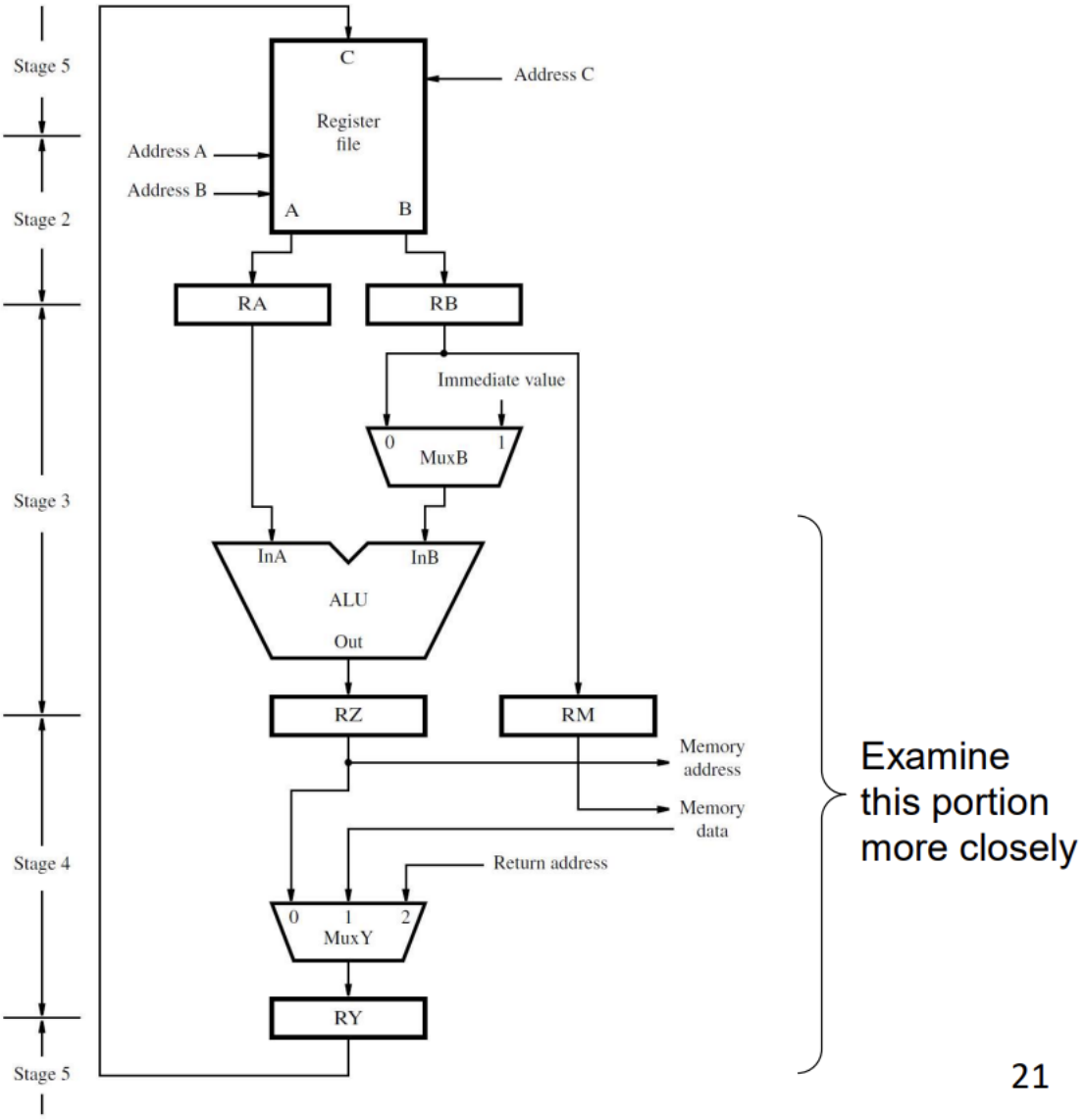
7.2 Hardware Datapath

Questions regarding **MuxPC**, **MuxY**, and **Register File inputs** require an understanding of how data is routed.

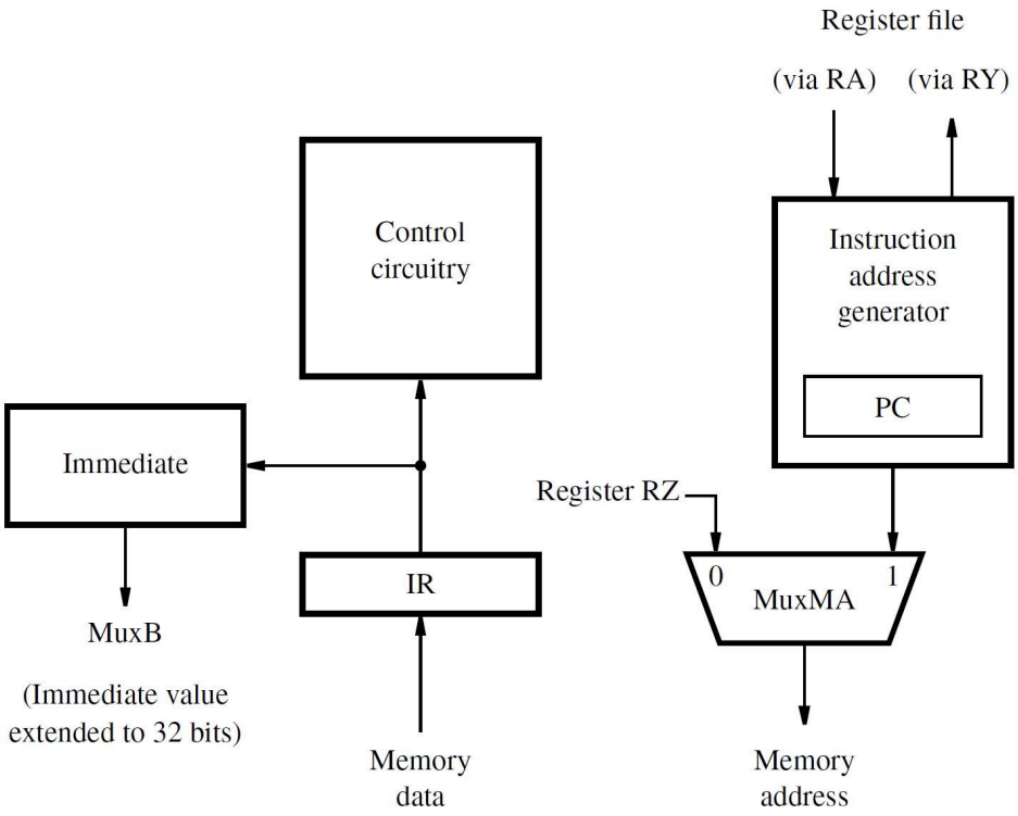
The figure below is often referred to as the **datapath**.

MuxY inputs are:

- RZ (ALU results)
- Memory Data (for Loads)
- Return Address (for calls/interrupts)



The organization of the **instruction fetch** section of the processor is in the figure below.

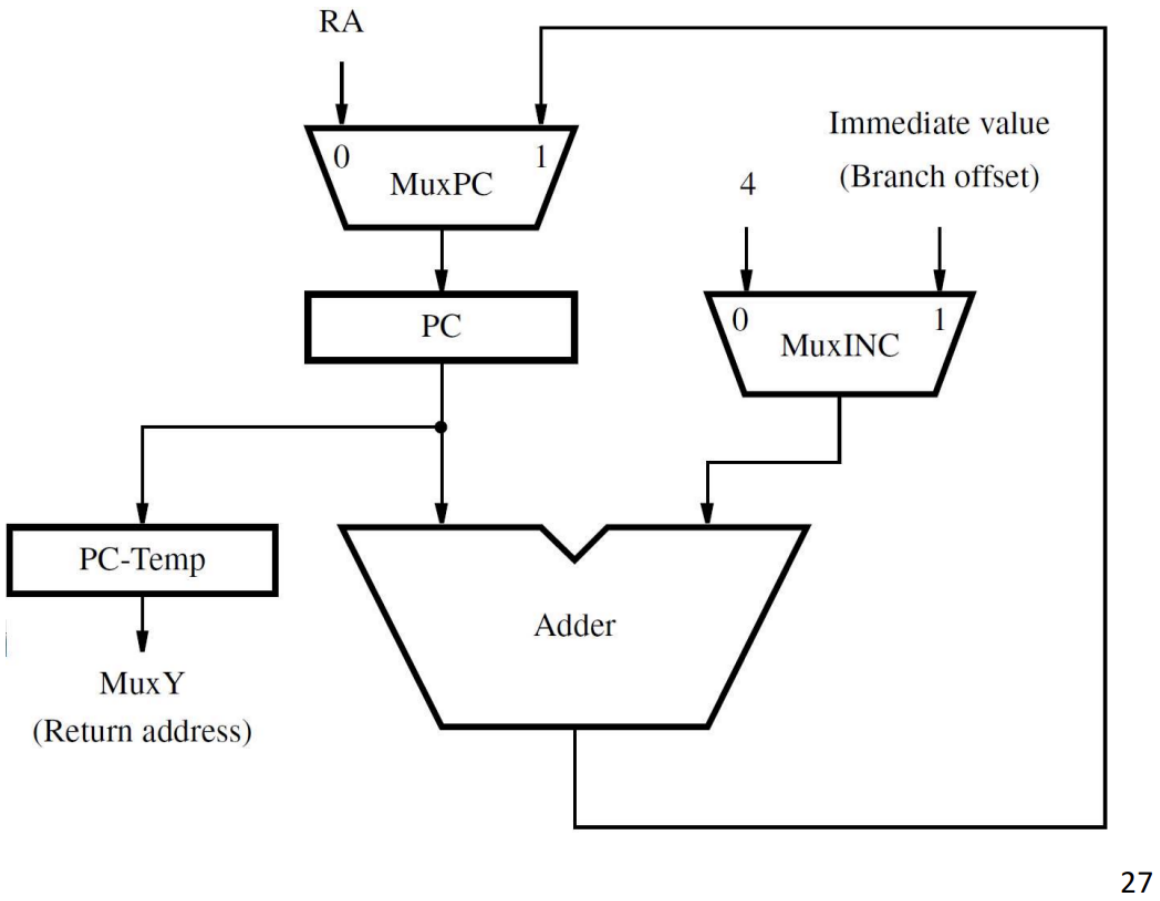


25

The **instruction address generator circuit** is shown below.

MuxPC inputs are:

- current incremented $PC + 4$ (for straight-line code)
- Branch Offset (for jumps)
- RA (for returns)



7.3 Control Signals

Exam questions ask you to write logic equations for control signals using the timing steps ($T1$–$T5$) and instruction signals, where Tn is allocated for Step n

RF_Write: You only write to the register file in Step 5 for instructions that produce a result

$$RF\_Write = T5(ALU + Load + Call)$$

Mem_Read: You read memory during Fetch and during the Memory stage of a Load

$$Mem\_Read = T1 + (T4 \cdot Load)$$

IR_en: The Instruction Register is only updated during the Fetch phase when the memory transfer is complete.

$$IR\_en = T1 \cdot MFC$$

PC_en: The Program Counter is updated after a fetch ($T1$) or after a branch/jump ($T3$).

$$PC\_en = (T1 \cdot MFC) + (T3 \cdot Branch)$$

7.4 Execution Tables

To solve the "Complete the Table" problems, remember that **inter-stage registers** (RA, RB, RZ, RY) hold the output of a stage to be used in the *next* cycle.

Example: `Subtract R5, R4, R3` (where R4=0x8765, R3=0x4321):

Cycle 2 (Decode): RA and RB are loaded. $RA = 0x8765, RB = 0x4321$.

Cycle 3 (Execute): The ALU subtracts them. $RZ = 0x8765 - 0x4321 = 0x4444$

Cycle 4 (Memory): RZ is passed to RY. $RY = 0x4444$.

Cycle 5 (Writeback): The value in RY is written to R5.