Very Large Scale Integration II - VLSI II

Instruction Set Architecture

Prof. Dr. Berna Örs Yalçın

ITU VLSI Laboratories

Istanbul Technical University

Topics for today

- Introduction to Processors
- Instruction Set Architecture

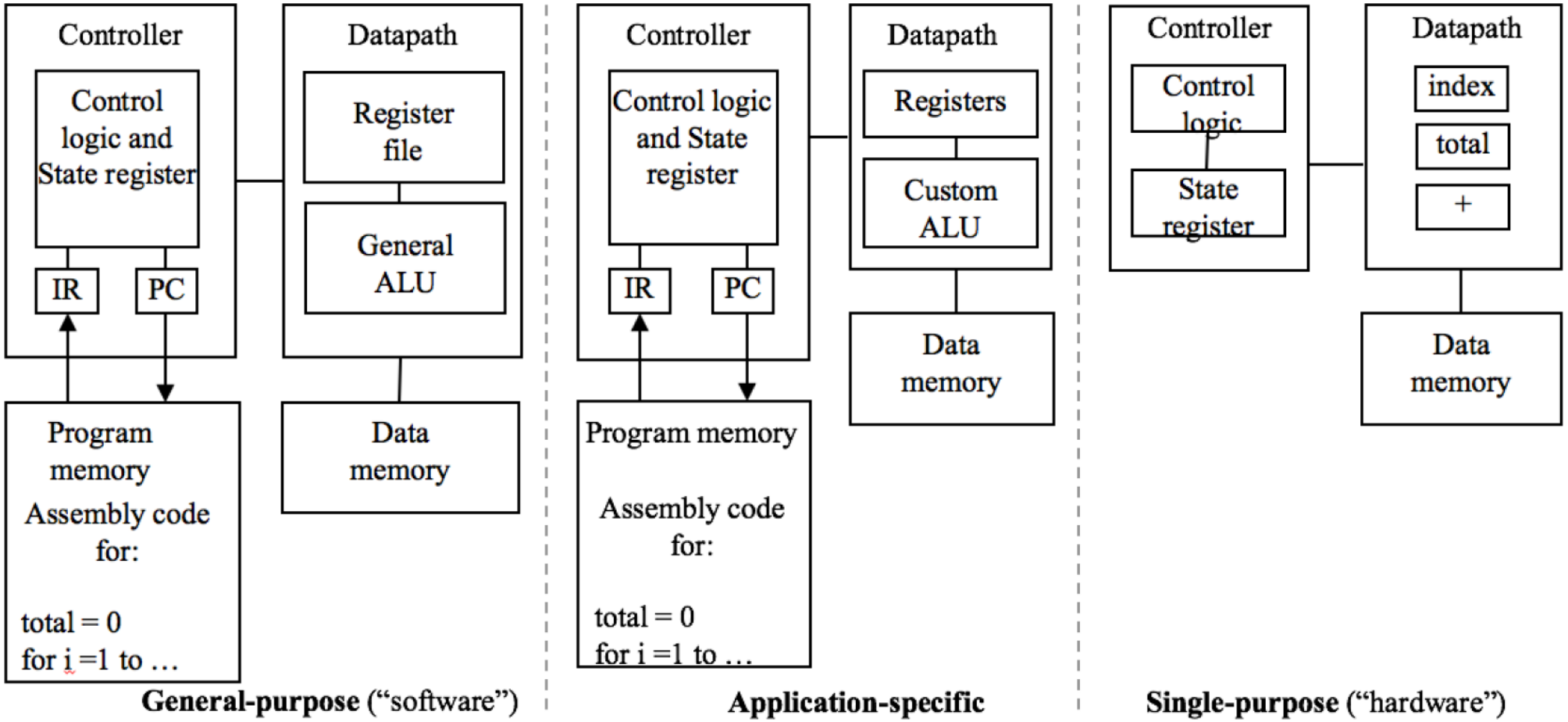
INTRODUCTION

Processors

- A processor is a digital circuit designed to perform computational tasks.
- A processor basically consists of:
  - **Datapath**: storing and manipulating data (ALU, Registers)
  - **Controller**: controlling movement of data through the datapath (Program Counter, Instruction Decoder)
  - **Memory:** while registers allow short term storage requirements, memory allow medium and long-term information-storage requirements.
    - Primary Memory: RAM, ROM
    - Secondary Memory: Hard Disk, Flash Disk

Processors

- **General-purpose processors** can carry out a wide variety of general computational tasks. (Intel 8051)
- **Application-specific processors** are optimized for a particular class of applications having common characteristics. (DSPs, GPUs)
- **Single-purpose processor** can only carry out particular computational task. (ASICs, co-processors like FFT, Channel Encoder/Decoders)



- **IR:** Instruction register, **PC:** Program Counter

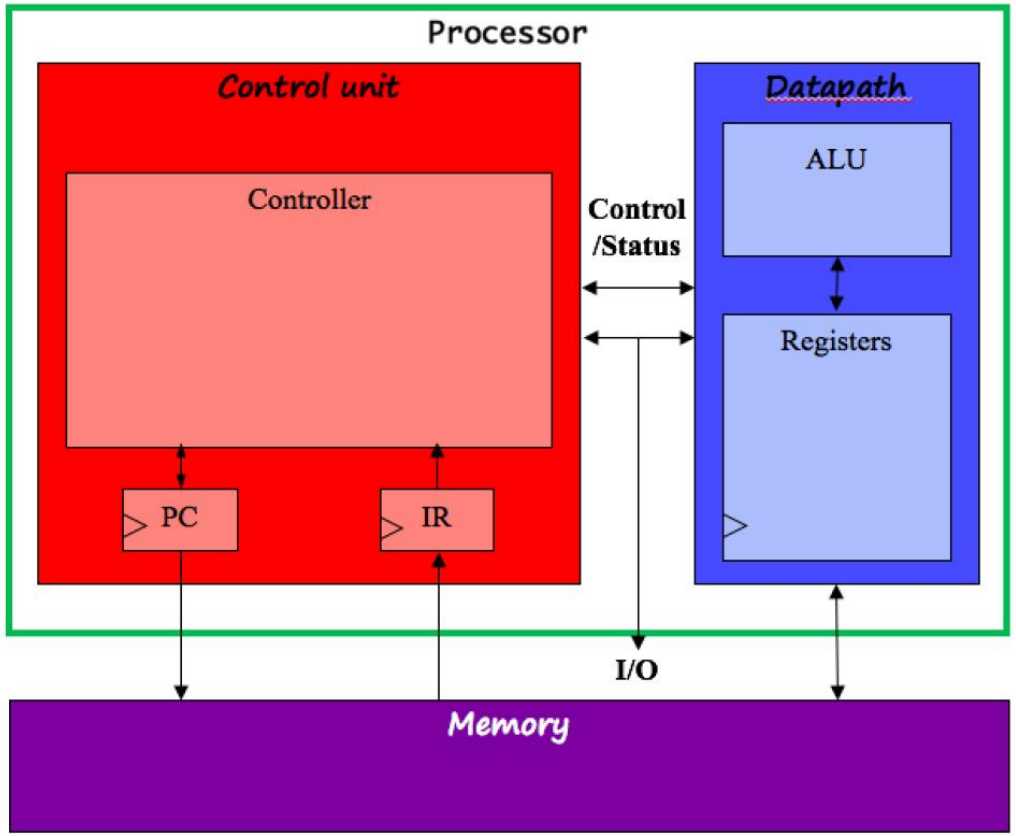
General-Purpose Processors

- They are designed for a variety of computation tasks (general tasks).
- Their datapath is general not application-specific.
- Control unit doesn't "store the algorithm", the algorithm is "programmed" into the memory. (reprogrammable)

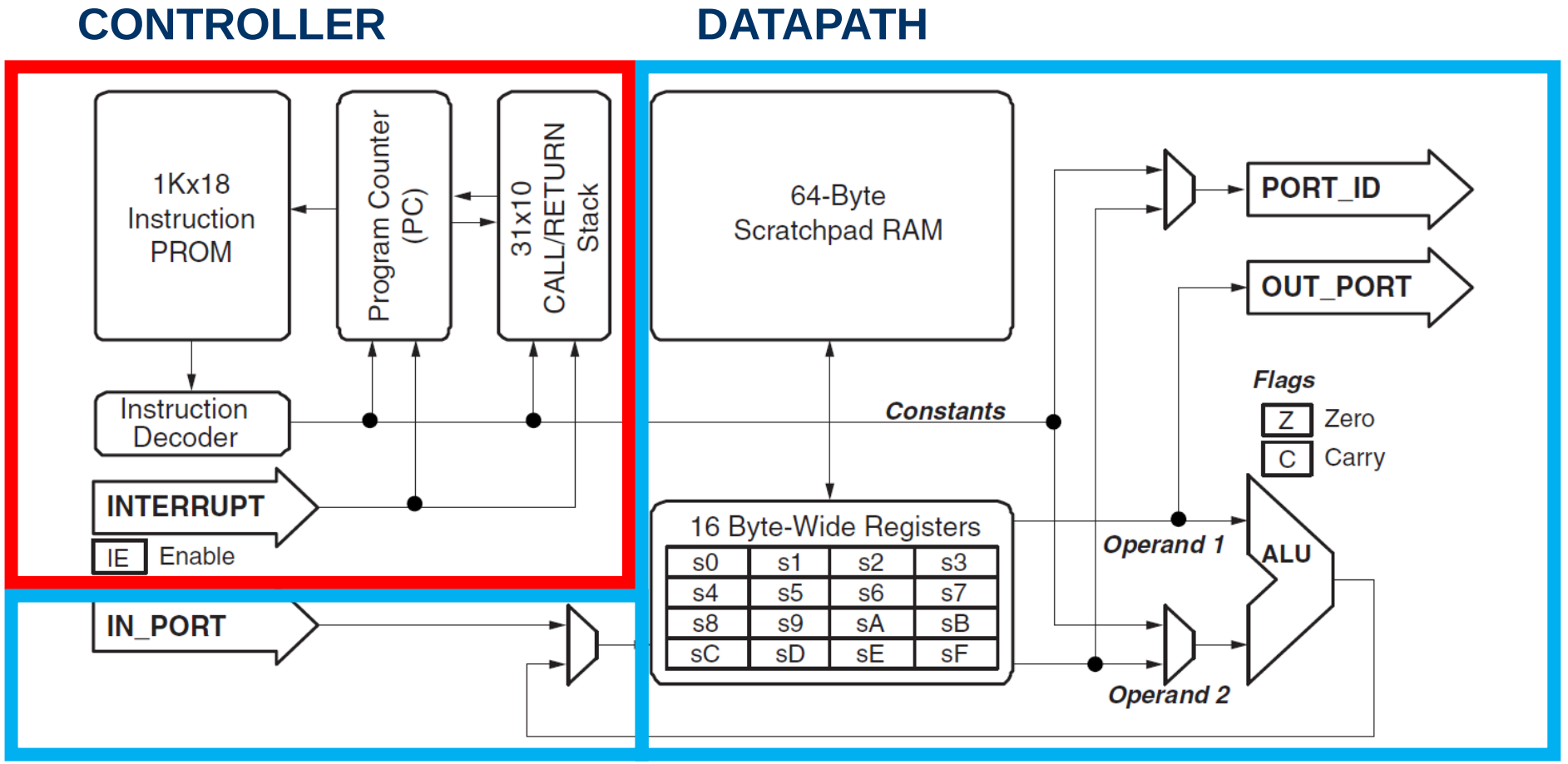
- Low unit cost, mass production.
- Low NRE (Non-Recurring Engineering) cost for embedded system designs,

General-Purpose Processors

- **Datapath Unit:** consists of circuitry for transforming data and for storing temporary data.
- **Control Unit:** consists of circuitry for retrieving program instructions and for moving data from and through the datapath.
- **N-bit processor:** N-bit ALU, registers, buses, memory data interface.



Picoblaze 8-bit Microprocessor



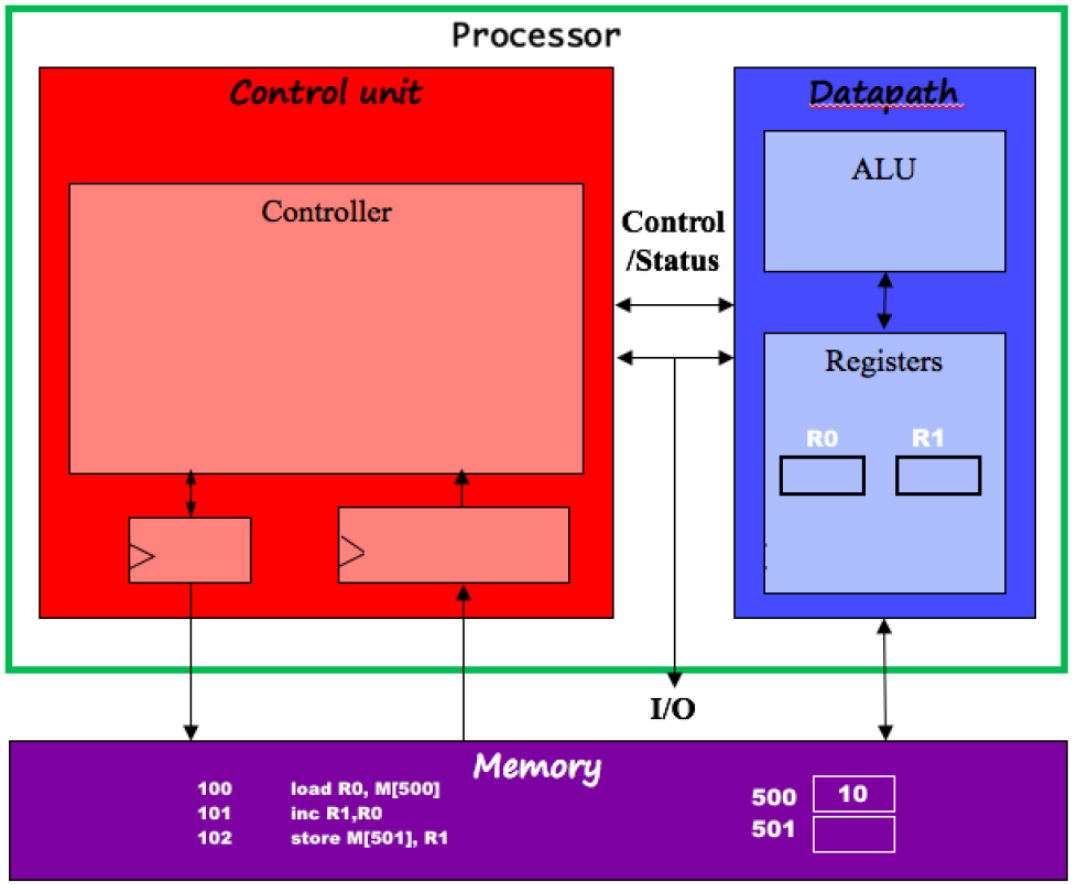
Operations & Instruction Cycle

- **Datapath Operations:**
  - Load: Read data
  - Execution (ALU): Perform computations
  - Store : Write output data (temporary place)
- **Control Unit Operations:**
  - Fetch : Get the instruction
  - Decode : Determine what the instruction means

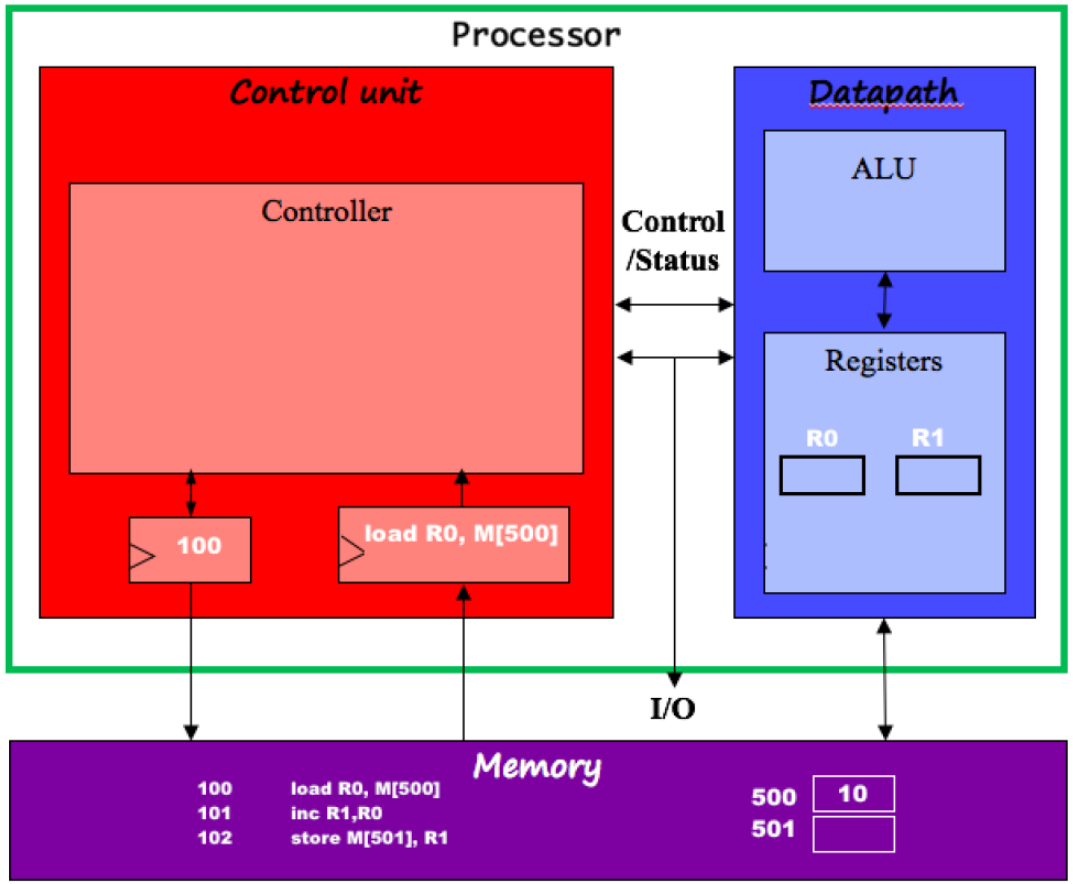
Operations & Instruction Cycle

- **Instruction Cycle:**
  - Fetch : Get the instruction
  - Decode : Determine what the instruction means
  - Fetch data : Move data from memory to datapath register
  - Execute : Move data through the ALU
  - Store : Store results

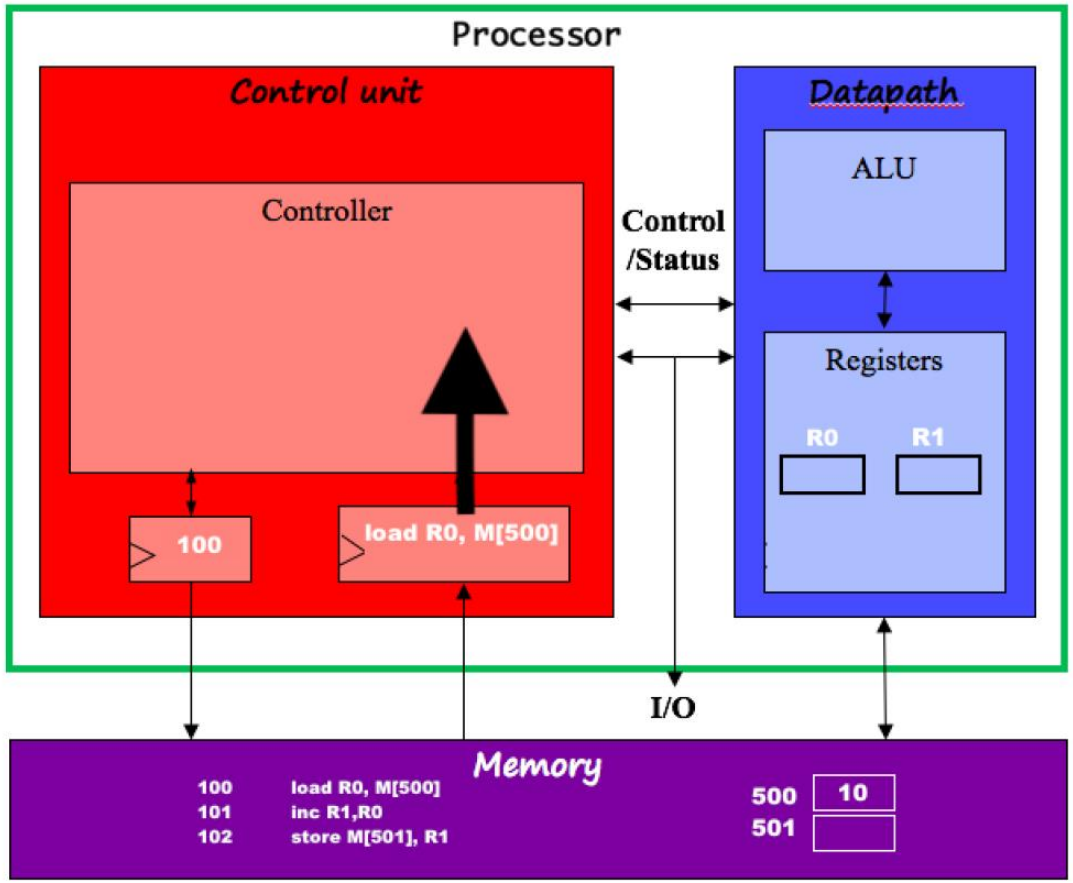
Instruction Cycle



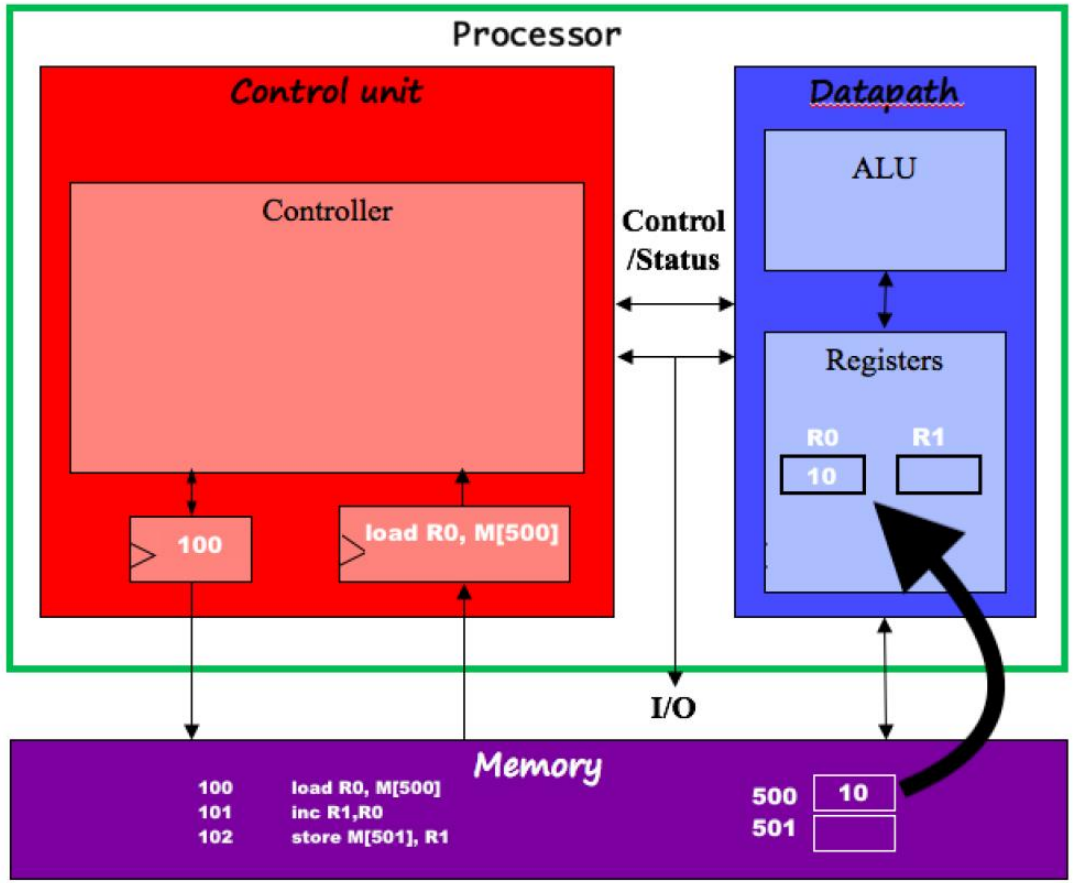
Fetch:


Processor
Control unit
Controller
100
load R0, M[500]
Control
/Status
Datapath
ALU
Registers
R0
R1
I/O
Memory
100 load R0, M[500]
101 inc R1,R0
102 store M[501], R1
500
10
501

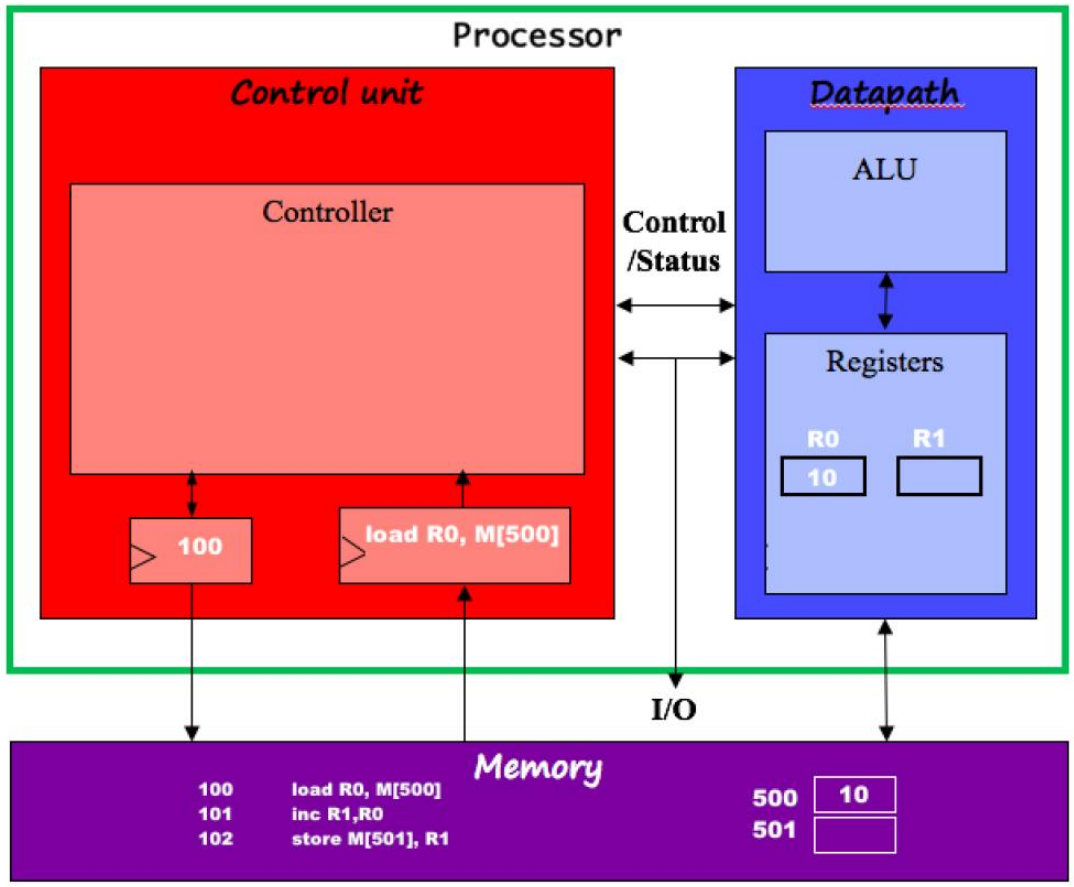
Decode:


Processor
Control unit
Controller
100
load R0, M[500]
Control
/Status
Datapath
ALU
Registers
R0
R1
I/O
Memory
100 load R0, M[500]
101 inc R1,R0
102 store M[501], R1
500
10
501

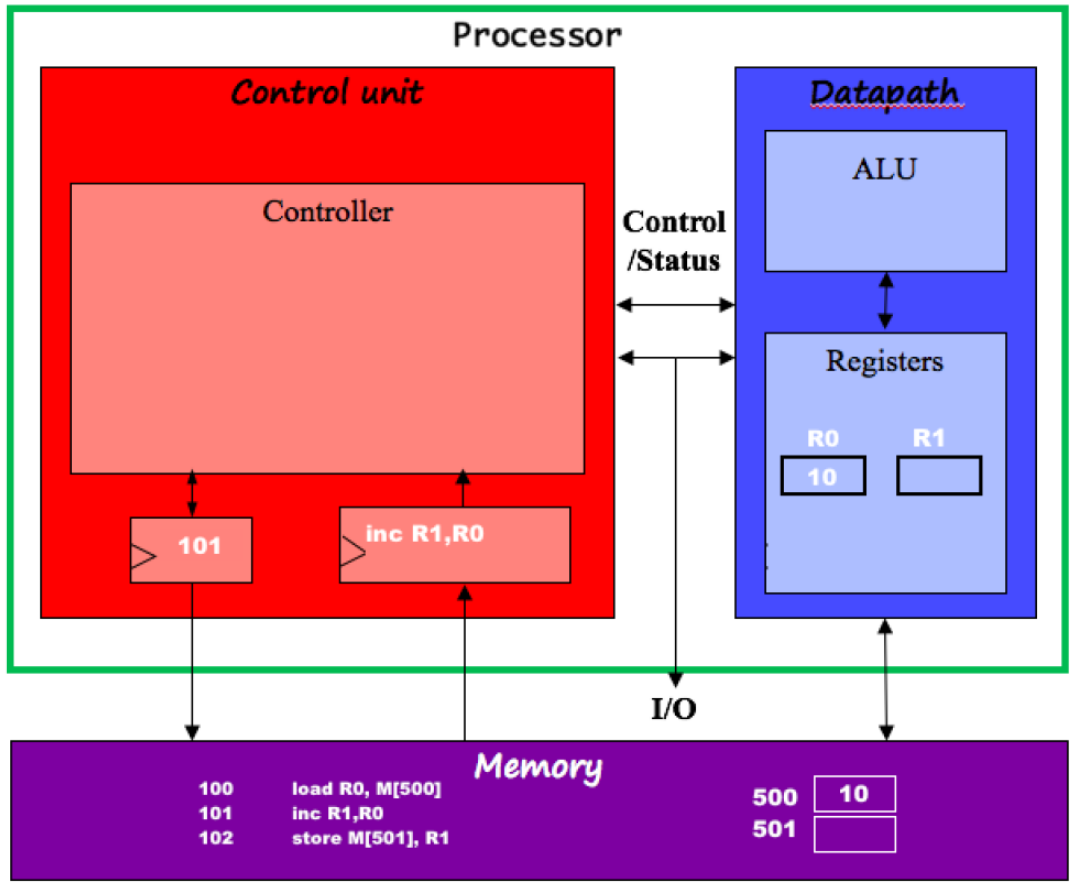
Fetch Operand:



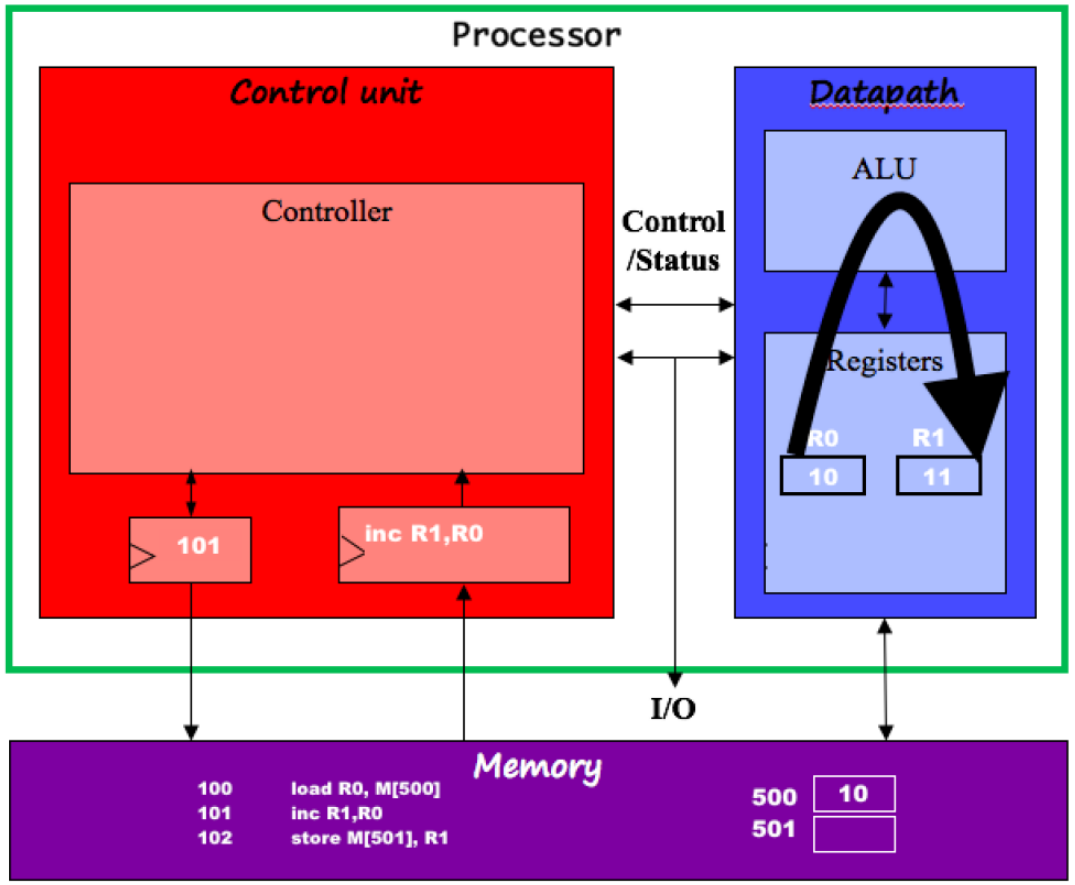
Execute & Store:



Fetch & Decode & Fetch Operand:



Execute & Store:



Characteristics/Performance Metrics

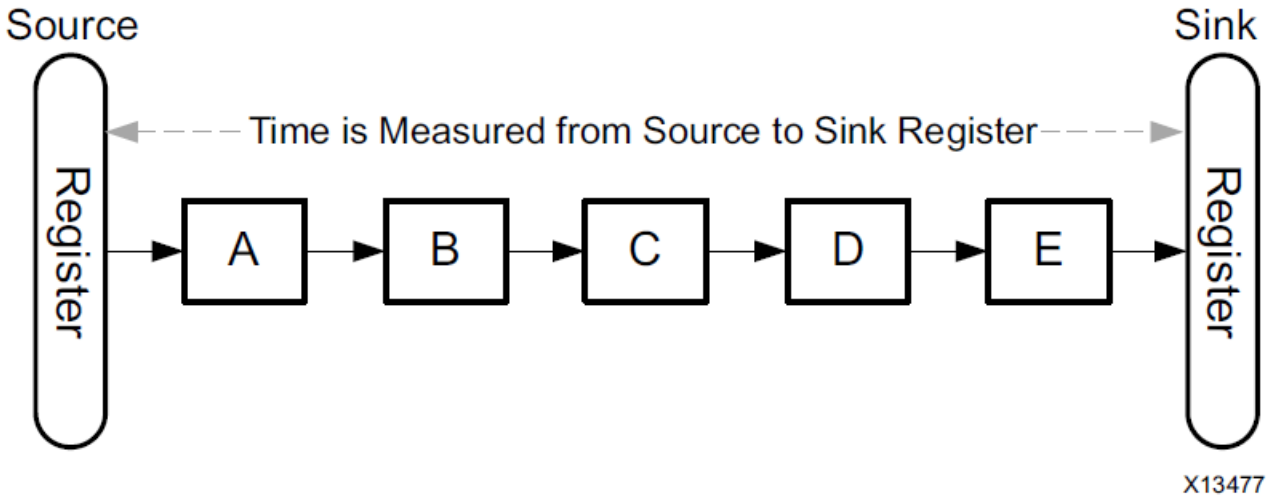
- **Word Size :** Number of bits that can be processed by a processor in a single instruction.
- **Instruction Set:** Basic set of machine level instructions.
- **Memory Architecture:** Determines the upper bound on achievable performance.
- **Clock Speed:** Limited by CMOS technology.
- **Latency:** Latency is the number of clock cycles it takes to complete an instruction or set of instructions. (Time between task start and end)
- **Throughput:** Determine overall performance (Tasks per second)
- **Power Consumption:** Determine by CMOS tech. (size of transistors), clock speed.
- **Size :** Determine by CMOS tech. and layout design.

Improving Performance

- **Faster clock:** There is a physical limit.
- **Pipelining:** Allows to avoid data dependencies by slicing up instructions into stages.
- **Parallelism:** Multiple ALUs to support more than one instruction stream.

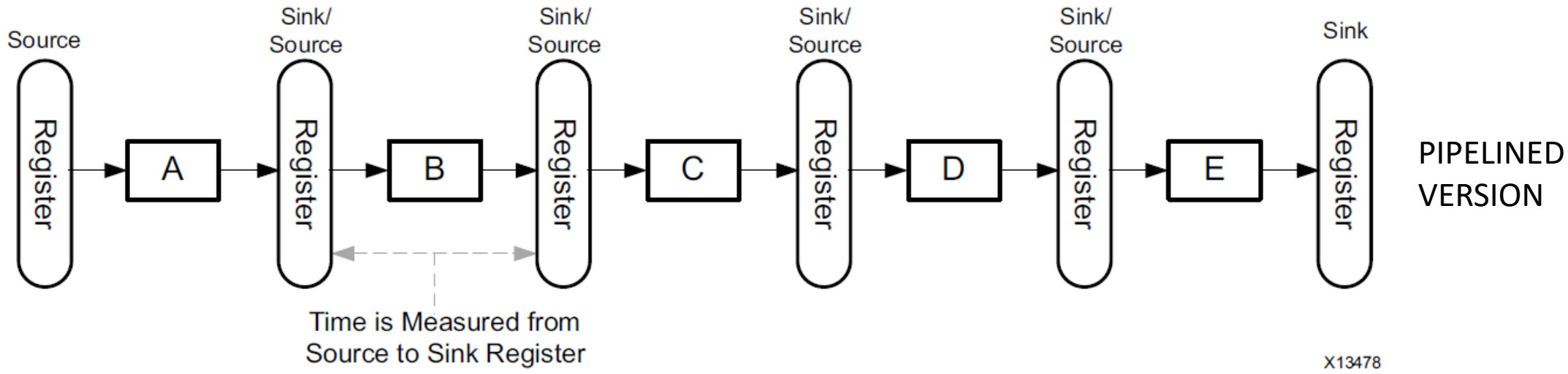
- By using pipelining, it is possible to increase clock frequency and throughput.

Pipelining



- LATENCY = 1 CLOCK CYCLE
- MAX_CLK_FREQ = 1 / (A + B + C + D + E + F)

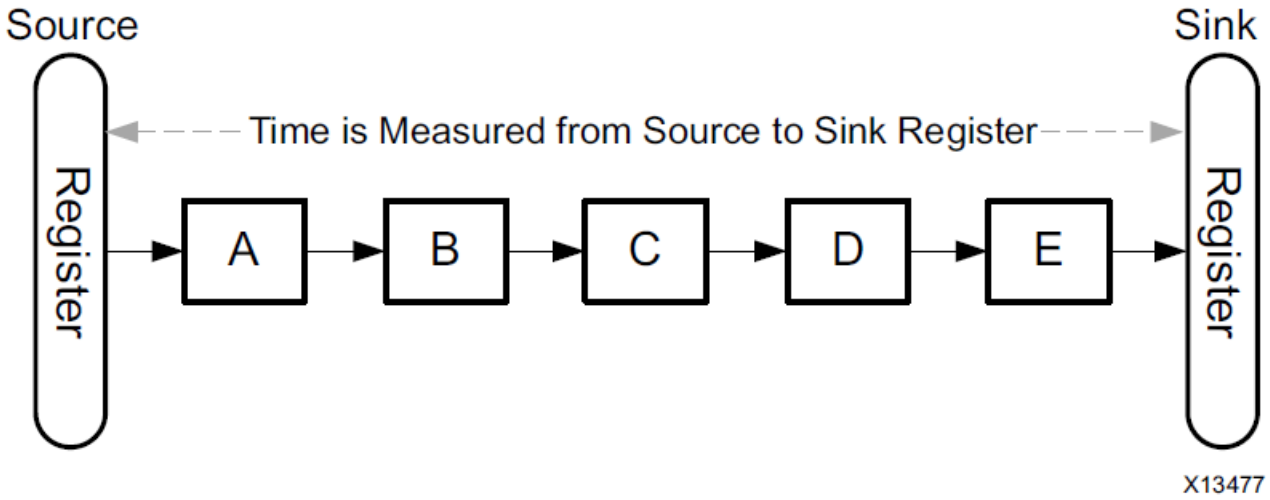
Pipelining



PIPELINED VERSION

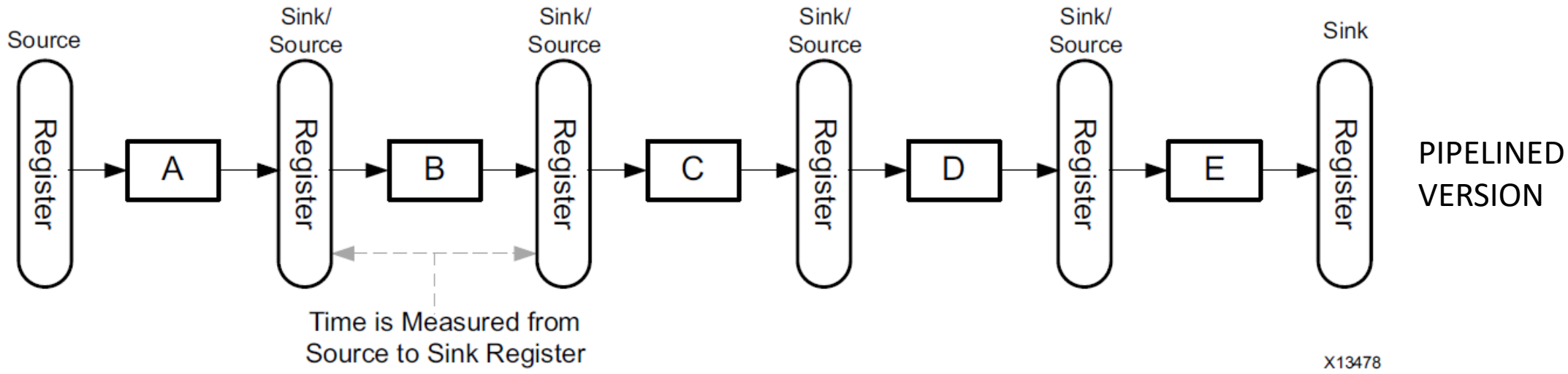
- LATENCY = 5 CLOCK CYCLE
- MAX_CLK_FREQ = max(1/A, 1/B, 1/C, 1/D, 1/E, 1/F)

Pipelining



- Let’s say latency is 10 ns.
- Processing a sample requires 10 ns

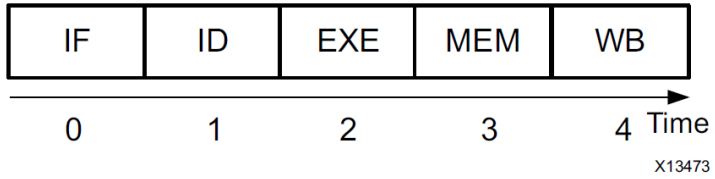
Pipelining



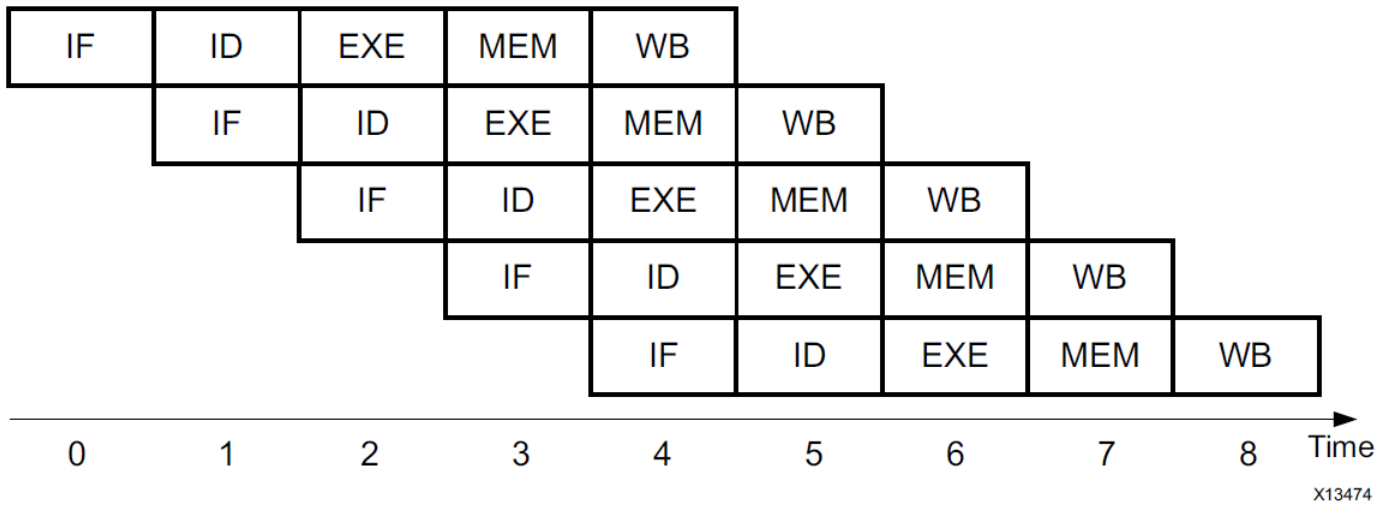
PIPELINED VERSION

- Let's say latency is 2 ns.
- Processing a sample requires 2 ns.

Pipelining



IF : Instruction Fetch
ID : Instruction Decode
EXE : Execute
MEM: Memory Operations (Fetch Data)
WB : Write Back (Store result)



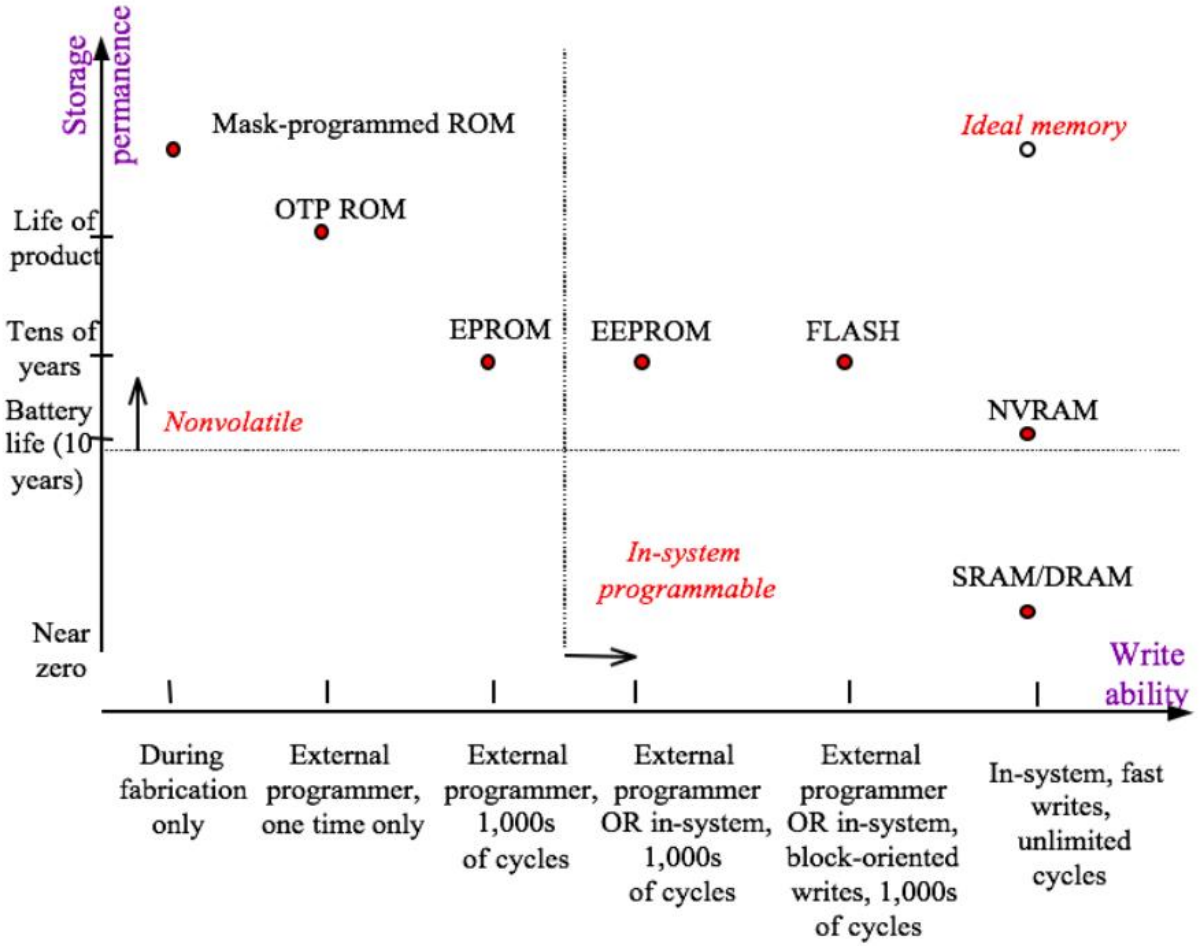
Pipelining of instruction cycle

Memory

- Traditional ROM/RAM distinctions:
  - ROM read only, data **can** be stored without power
  - RAM read and write, data **can not** be stored without power
- Traditional distinctions are blurred
  - Advanced ROMs can be written e.g., EEPROM
  - Advanced RAMs can hold bits without power e.g., NVRAM

Memory

- There is trade-off between the Write ability and storage permanence (life)
- Write ability: write speed of a memory.
- Storage permanence : ability of memory to hold stored bits after they are written.

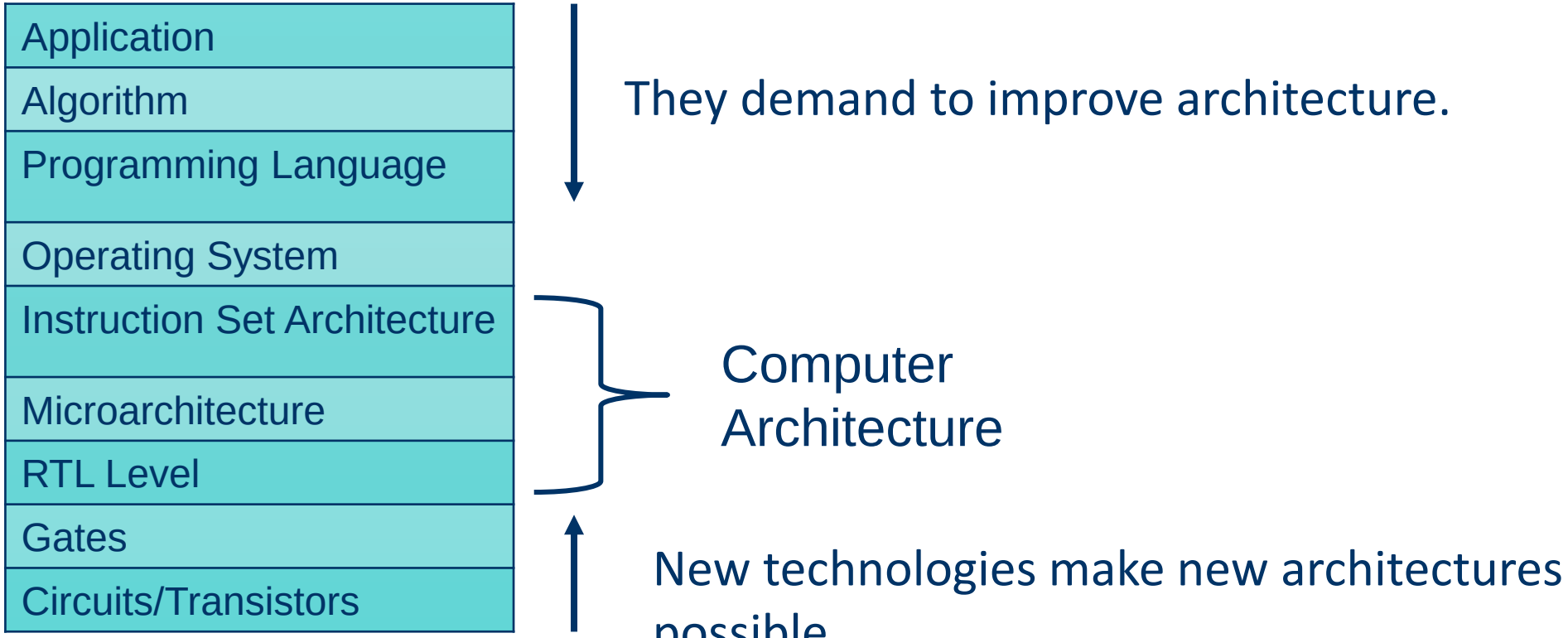


Memory

- Ranges of write ability:
  - High end [RAM]
  - Middle range [EEPROM]
  - Lower range [EPROM, OTP ROM]
  - Low end [Mask-programmed ROM]
- Range of storage permanence
  - High end [mask-programmed ROM]
  - Middle range [NVRAM]
  - Lower range [SRAM]
  - Low end [DRAM]

INSTRUCTION SET ARCHITECTURE

Abstraction in Modern Computer Systems



Computer Architecture

- **Instruction Set Architecture:** What to do by a processor?
- **Micro Architecture:** How to implement those instructions?

- Micro Architecture is also called '**Computer Organization'** or just **'Organization'**.
- Instruction Set Architecture can be abbreviated to **'Architecture'**.

Instruction Set Architecture

- **Instruction Set Architecture (ISA)** specifies the behavior of machine code of a processor. It is an abstract definition.
- Realization of an ISA is called **an implementation**.
- ISA provides **compatibility** between different implementations.
- Processors having same ISA can run the same executable code!

ISA vs Micro Architecture

- **Instruction Set Architecture** specifies:
  - Operations (Instructions and how they work)
  - Memory and Register (Does it have memory and registers, purpose of registers)
  - Data types (Which data types are available)
  - Inputs and Outputs
  - Interrupts
- **Micro Architecture** specifies:
  - How to implement ISA.
  - Tradeoffs: speed, energy and cost.
  - The performance.

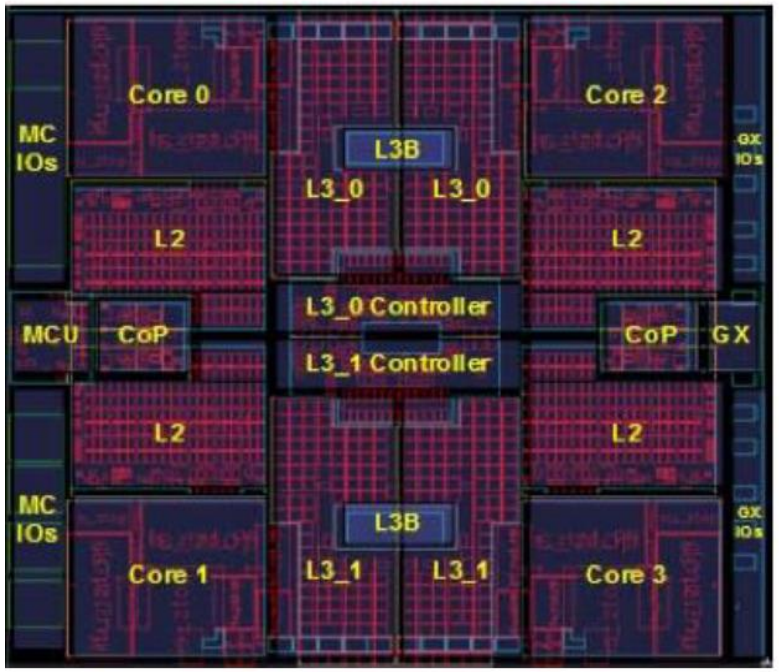
ISA vs Micro Architecture

- Different implementations of an ISA can differ in performance, cost and hardware structure.
- **IBM 360 (1964):** It is the first design that made a clear distinction between instruction set and micro architecture. This was the milestone.
- **IBM 360** instruction set architecture (ISA) completely hid the underlying technological differences between various models.

IBM 360

	Model 30	. . .	*Model 70*
Storage	8K - 64 KB		256K - 512 KB
Datapath	8-bit		64-bit
Circuit Delay	30 nsec/level		5 nsec/level
Local Store	Main Store		Transistor Registers
Control Store	Read only 1μsec		Conventional circuits

IBM 360 47 years later:



[IBM, Kevin Shum, HotChips, 2010]
Image Credit: IBM
Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

- 5.2 GHz in IBM 45nm PD-SOI CMOS technology
- 1.4 billion transistors in 512 mm^2
- 64-bit virtual addressing
  - original S/360 was 24-bit, and S/370 was 31-bit extension
- Quad-core design
- Three-issue out-of-order superscalar pipeline
- Out-of-order memory accesses
- Redundant datapaths
  - every instruction performed in two parallel datapaths and results compared
- 64KB L1 I-cache, 128KB L1 D-cache on-chip
- 1.5MB private L2 unified cache per core, on-chip
- On-Chip 24MB eDRAM L3 cache
- Scales to 96-core multiprocessor with 768MB of shared L4 eDRAM

AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

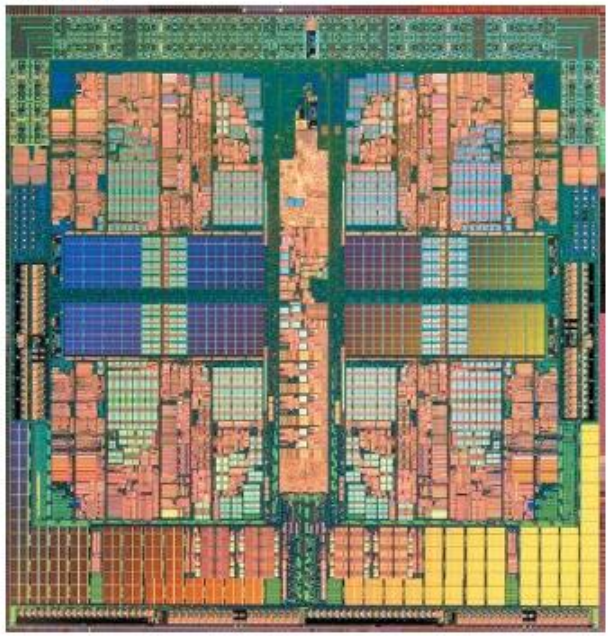


Image Credit: AMD

Intel Atom

- X86 Instruction Set
- Single Core
- 2W
- Decode 2 Instructions/Cycle/Core
- 32KB L1 I Cache, 24KB L1 D Cache
- 512KB L2 Cache
- In-order
- 1.6GHz

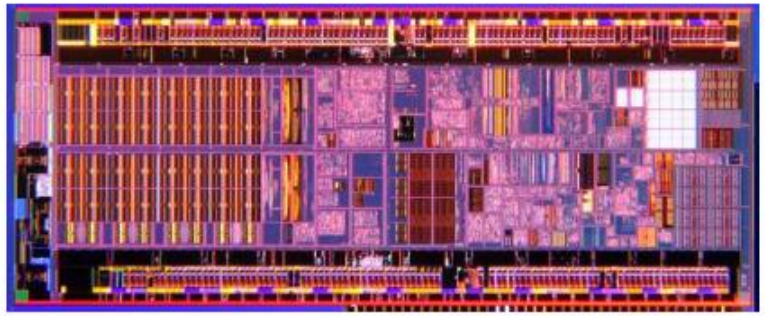


Image Credit: Intel

Instruction Set Architecture

- An ISA can be classified in different ways.
- By **Architectural Complexity:**
  - CISC (Complex Instruction Set Computer)
  - RISC (Reduced Instruction Set Computer)
- By **Instruction Level Parallelism**
  - VLIW (very long instruction word)
  - LIW (long instruction word)
  - EPIC (explicitly parallel instruction computing)

Instruction Set Architecture

- **CISC (Complex Instruction Set Computer)**
  - Example: x86 ISA, Implementation: Intel Pentium
  - Make multiple operations in one instruction
  - ALU, memory and I/O operations in same instruction
  - Variable-lenght instructions
  - Require larger area and consumes more power!

Instruction Set Architecture

- **RISC (Reduced Instruction Set Computer)**
  - Simpler instructions than CISC
  - Fewer and general instructions
  - Requires less cycles to execute
  - Generally, fixed-length instructions
  - Generally, fixed-length registers
  - Easy to develop control logic!
  - Easy to design and optimise compiler!
  - Lower area, lower power, but also low performance! (?)

CISC vs RISC

- The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible.

- For example:
  - `MULT 0x100, 0x104`

- This instruction loads the two values from memory (0x100, 0x104) into separate registers, multiplies the operands in the execution unit, stores the product in the appropriate register, and then store the result back in the given location. Thus, the entire task of multiplying two numbers can be completed with one instruction.

CISC vs RISC

- RISC processors only use simple instructions that can be executed within one clock cycle.
- Thus, the "MULT" command could be divided into four separate commands:
  - `LOAD A, 0x100`
  - `LOAD B, 0x104`
  - `MUL A, B`
  - `STORE 0x100, A`

- "LOAD," moves data from the memory to a register (A,B), "MUL," finds the product of two operands, and "STORE," moves data from a register to a memory location.

CISC vs RISC

- **CISC:**
  - Emphasis on hardware
  - Includes multi-clock complex instructions
  - Small code sizes, high cycles per instruction
- RISC:
  - Emphasis on software
  - Single-clock, reduced instruction only
  - Large code sizes, low cycles per instruction.

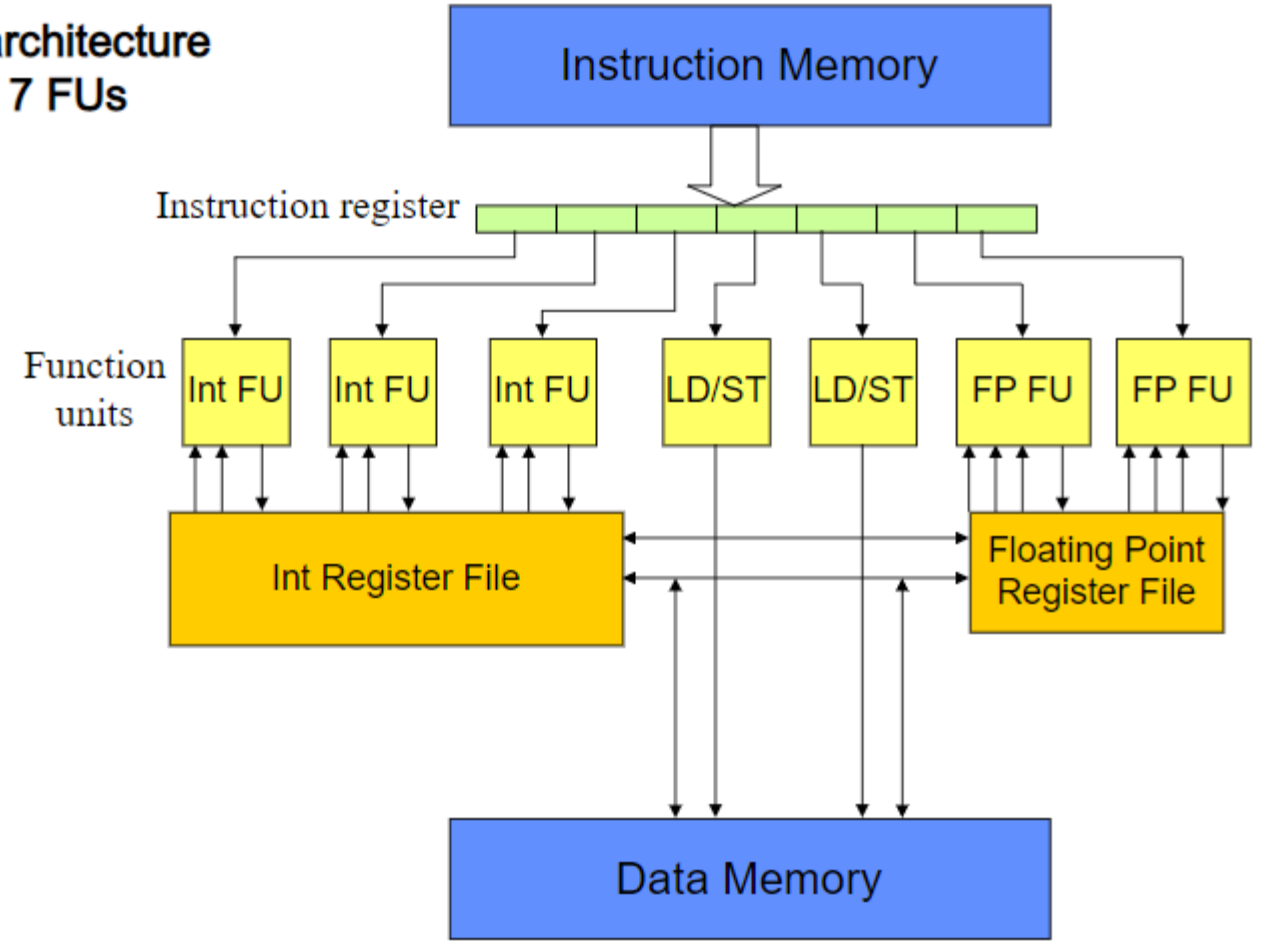
- The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction.
- RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

Instruction Set Architecture

- **VLIW (very long instruction word)**
  - Designed to use the advanteges of Instruction Level Parallelism (ILP),
  - Executes many instructions in parallel,
  - High performance for some applications,
  - Compilers far more complex,
  - Long compilation time,
  - May not be proper for general purpose processing

VLIW: general concept

A VLIW architecture with 7 FUs



Instructions

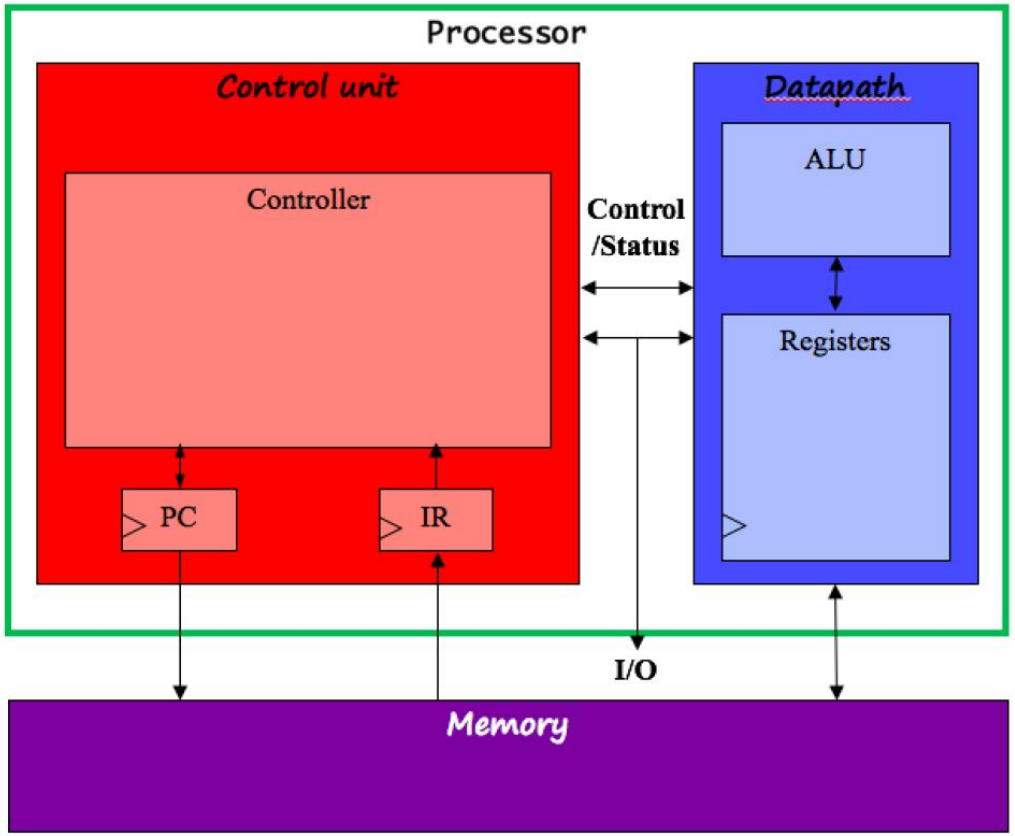
- A processor usually has a variety of instructions and multiple instruction formats.
- General Types of Instructions
  - ALU Instructions (Data operations in ALU)
  - Memory Instructions (Read or write data to memory)
  - Control Instructions (Branching, subroutines, interrupts)

Instructions

- ***Machine Language*** is the binary language in which instructions are defined and stored in memory.
- ***Assembly Language*** is a symbolic language that replaces binary codes with symbolic names.
- The logical structure of processors is normally described in assembly-language reference manuals.
- The control unit decode each instruction and provide the control signals needed to process it.

Instructions

- The **program counter (PC)** keeps track of the instructions in the program stored in memory. The PC holds the address of the instruction to be executed.
- The **instruction register (IR)** holds the instruction to be decoded.



Instruction Format

- Instructions are also kept as binary numbers.
- The bits are divided into groups called **fields**. The following are typical fields found in instruction formats:
  - **Opcode** field, which specifies the operation to be performed.
  - **Address** field, which provides either a memory address or an address of a register.
  - **Mode** field, which specifies the way the address field is to be interpreted (register or memory address).

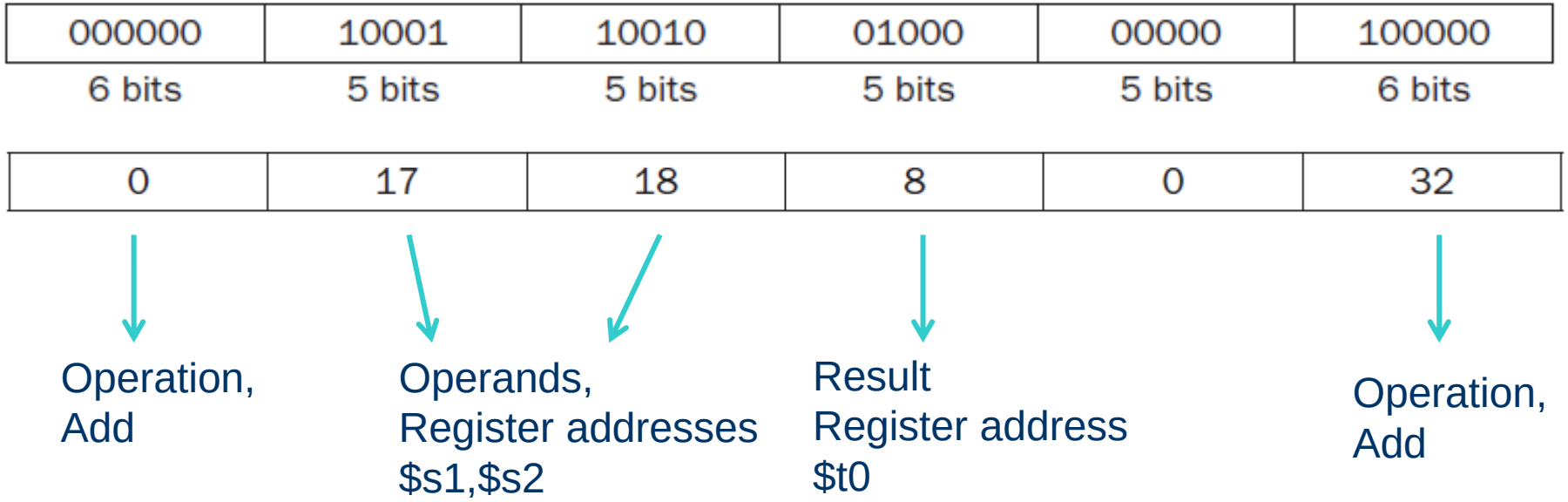
- **MIPS (Microprocessor without Interlocked Pipelined Stages)**, which is a RISC type ISA, will be covered.
- We will also use **MIPS assembly language** notation in later chapters.

Instruction Format

- An arithmetic instruction:

```
add $t0,$s1,$s2
```

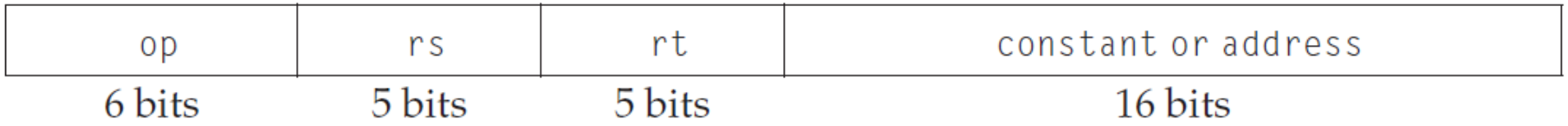
- Machine Code:



Instruction Format

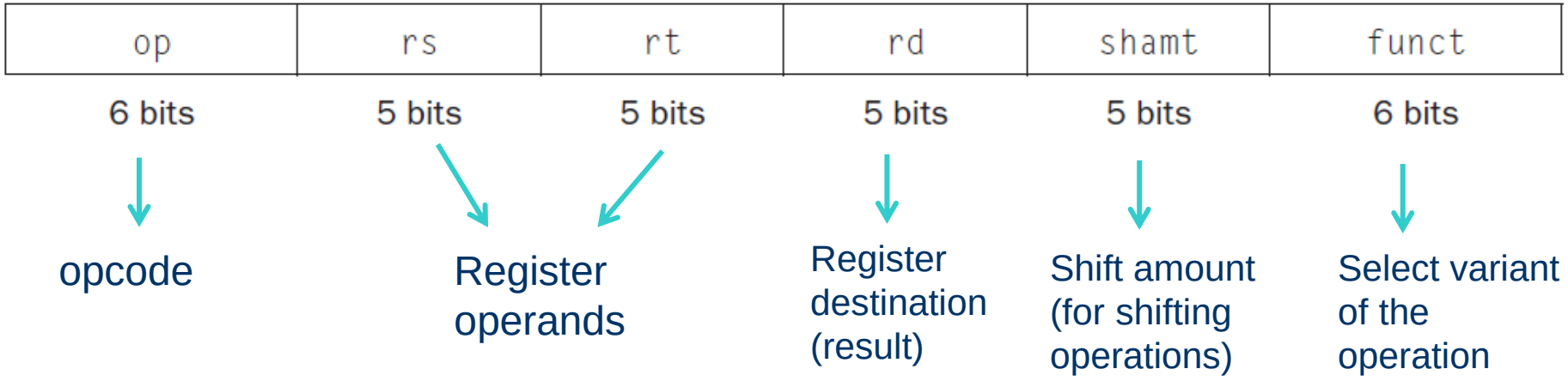
- All instructions should be the same length,
- Multiple formats complicate the hardware, they should be similar.
- Format for registers:

- Format for constants:



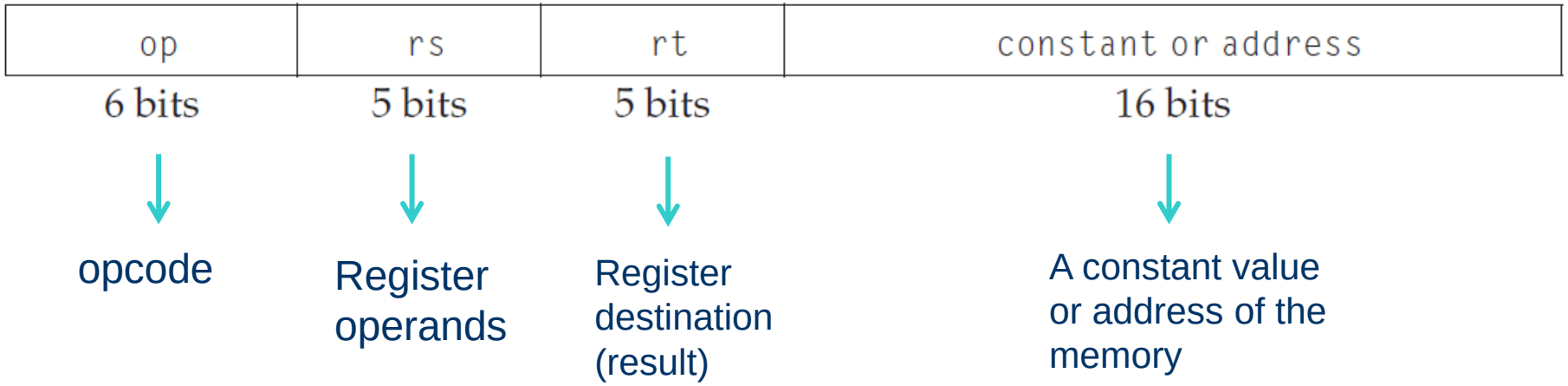
Instruction Format

- R-type Format (Register type)



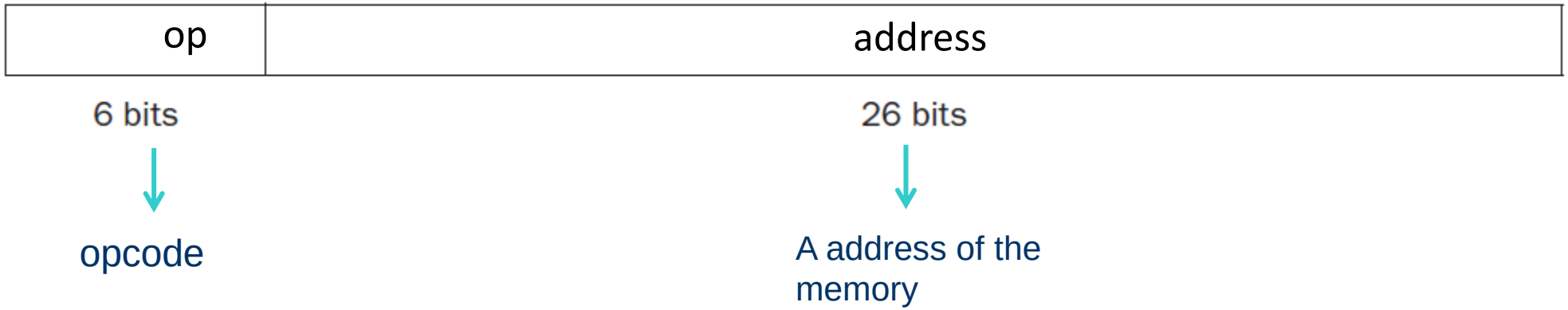
Instruction Format

- I-type Format (Immediate type)



Instruction Format

- J-type Format (Jump type)



- J-type format is used for jump instructions

Instruction Format

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Arithmetic Operations

- Every processor must be able to perform arithmetic operations.
- Each MIPS arithmetic instruction performs only one operation and must always have exactly three variables (addresses actually).

```
add a, b, c     # The sum of b and c is placed in a.
add a, a, d     # The sum of b, c, and d is now in a.
add a, a, e     # The sum of b, c, d, and e is now in a.
```

- It takes 3 instructions to take the sum of 4 variables.
- Requiring every instruction to have exactly three operands allows to keep the hardware simple.

Arithmetic Operations

- Compiling a C code:

```
f = (g + h) – (i + j);
```

- The compiler must break this statement into several assembly instructions

```
add t0, g,h     # temporary variable t0 contains g + h
add t1, i,j     # temporary variable t1 contains i + j
sub f ,t0,t1    # f gets t0 – t1, which is (g + h)–(i + j)
```

Registers

- The size of a register in the MIPS architecture is 32 bits and MIPS has 32 registers.
- The size 32 bits is referred as a ***word***.
- The reason for the limit of 32 registers:
  - A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel further.
- Arithmetic operations occur only on registers (except constants).

Registers

- Compiling a C code:

```
f = (g + h) – (i + j);
```

- Instead of variables we use registers:

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

Data Transfer Instructions

- MIPS also include data transfer instructions.
- To access a word in memory, the instruction must supply the memory address.
- Compiling a C code:

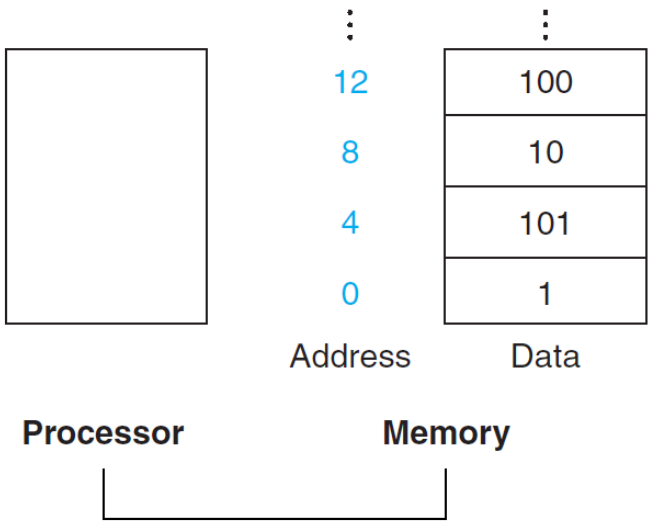
```
f = h + A[8];
```

- Assembly:

```
lw $t0, 8($s3)   # Temporary reg $t0 gets A[8]
                 # $s3 is the array address.
                 # 8th element of the array (offset).
```

Data Transfer Instructions

- Most architectures address individual bytes. In the previous example we address a complete word (32 bit – 4 byte)
- Therefore, the address of a word matches the address of one of the 4 bytes within the word.



Data Transfer Instructions

- Compiling a C code:

```
f = h + A[8];
```

- Assembly:

```
lw $t0, 32($s3)   # Temporary reg $t0 gets A[8]
add $t0,$s2,$t0   # Temporary reg $t0 gets h + A[8]
sw $t0,48($s3)    # Stores h + A[8] back into A[12]
```

Data Transfer Instructions

- Data accesses are faster if data is in registers instead of memory.
- Compiler tries to keep the most frequently used variables in registers and places the rest in memory.
- Constant operands occur immediately.
- We don’t have to load constants to registers, we can take them directly from memory.

```
addi $s3,$s3,4 # $s3 = $s3 + 4
```

Logical Operations

- Logical instructions perform binary operations on words stored in registers or memory.
- Logical instructions are bitwise operations.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Branch Instructions

- A branch provides fetching instructions from a different sequence of the memory.
- When a branch is taken, the program counter is set to the argument of the jump instruction.
- So, the next instruction becomes the instruction at that address in memory. Therefore, the flow of control changes.
- A branch instruction can be either an unconditional branch, which always results in branching, or a conditional branch, which may or may not cause branching depending on some condition.

Branch Instructions

- Compilers frequently create branches and labels where they do not appear in a high level programming language.
- Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages (instead of assembly).
- MIPS assembly language includes two decision-making instructions:

```
beq register1, register2, L1
bne register1, register2, L1
```

  - Go to the branch labeled L1 if the condition is satisfied.

Conditional Branch Instructions

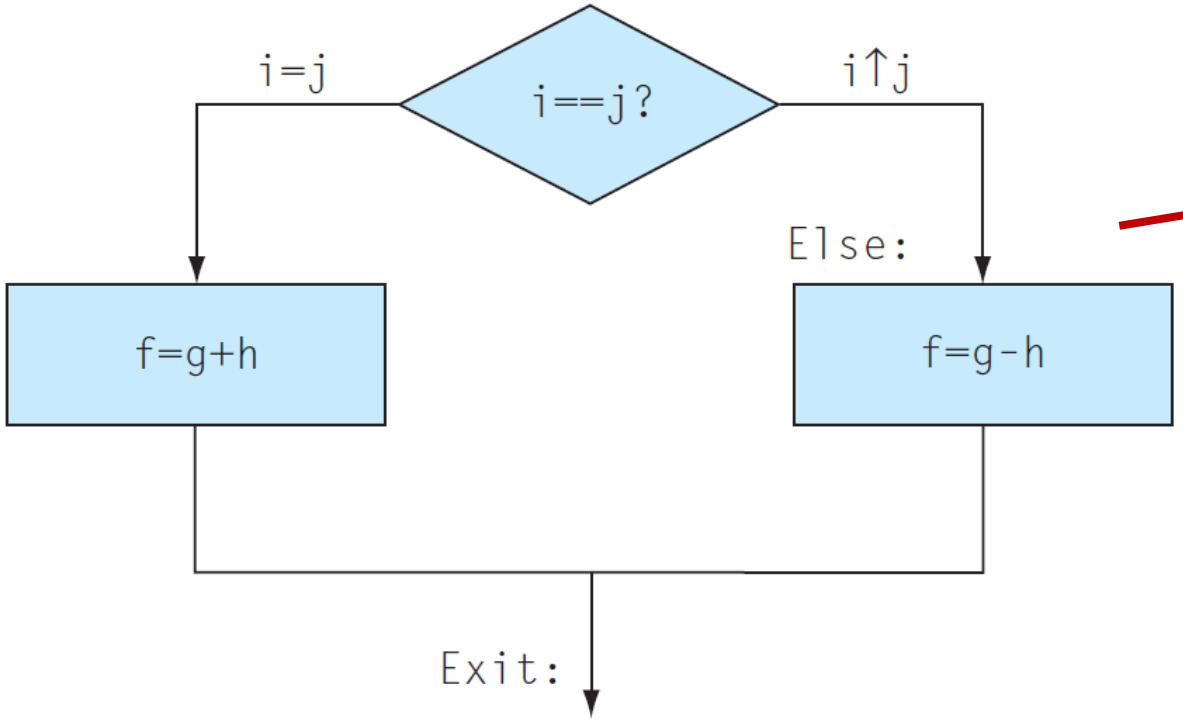
- Compiling a C code:

```
if (i == j) f = g + h;
else f = g - h;
```

- Assembly code:

```
      bne $s3,$s4, Else   # go to Else if i != j
      add $s0,$s1, $s2    # f = g + h (skipped if i != j)
      j Exit              # go to Exit (jump)
Else: sub $s0,$s1,$s2     # f = g - h (skipped if i = j)
Exit:
```

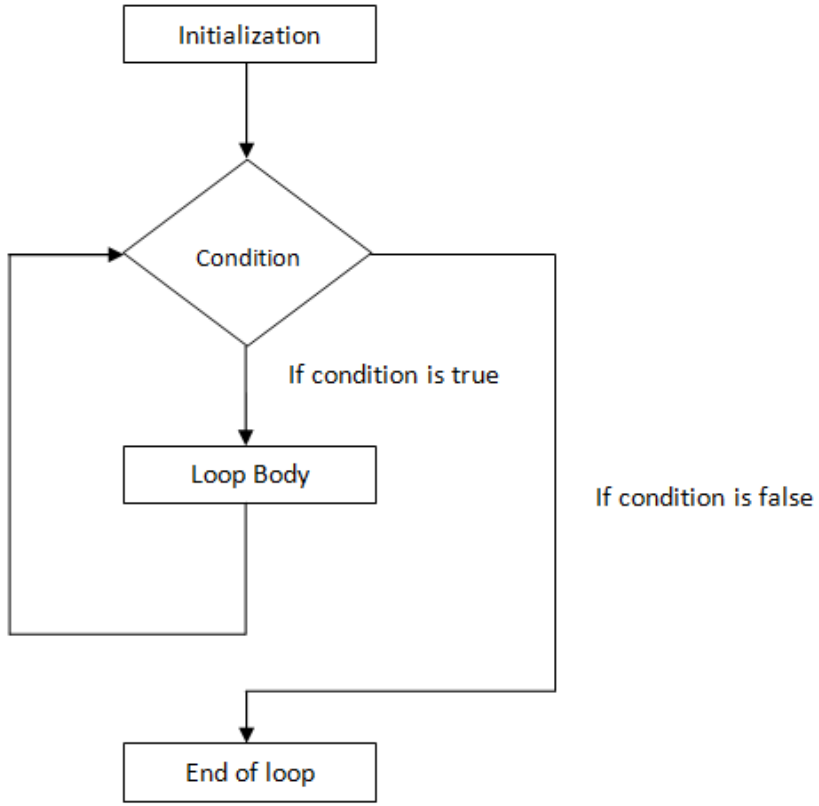
Conditional Branch Instructions



Branching:
Jump different address in the program memory.

Loops

- Branch instructions can also be used to implement loops.
- The iteration number is first loaded into a register which is subsequently manipulated and the iteration number is compared with a value in an another register or a constant.



Loops

- Compiling a C code:

```
while (save[i] == k) i += 1;
```

- Assembly code:

```
                                # save,i,k corresponds to s6,s3,s5
Loop: sll $t1,$s3,2             # Temp reg $t1 = 4 * i (addressing)
      add $t1,$t1,$s6           # $t1 = address of save[i]
      lw  $t0,0($t1)            # Temp reg $t0 = save[i]
      bne $t0,$s5, Exit         # go to Exit if save[i] != k
      add $s3,$s3,1             # i = i + 1
      j Loop                    # go to Loop
Exit:
```

Comparisons

- The test for equality or inequality is the most popular test, but sometimes it is useful to see if a variable is less than another variable (comparisons).

- Such comparisons in MIPS are:

```
slt $t0, $s3, $s4
```

  - register $t0 is set to 1 if the value in register $s3 is less than the value in register $s4
  - otherwise, register $t0 is set to 0.

Comparisons

- Constant operands are also be used in comparisons.

```
slti $t0,$s2,10        # $t0=1 if $s2<10
```

- MIPS compilers use the **slt**, **slti**, **beq**, **bne**, and the fixed value of 0 (always available by reading register **$zero**) to create all relative conditions:
  - equal, not equal,
  - less than, less than or equal,
  - greater than, greater than or equal.

Case/Switch Statement

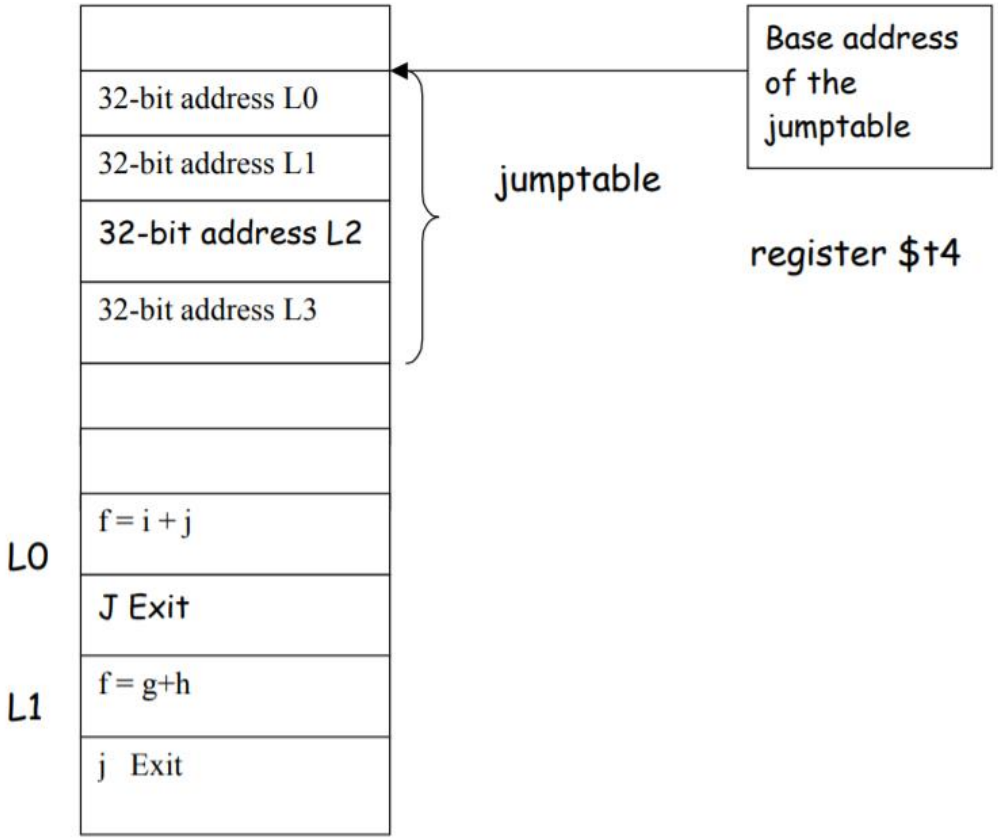
- Most programming languages have a case or switch statement that allows the programmer to select one of many alternatives depending on a single value.
- The simplest way to implement switch is via a sequence of conditional tests, turning the switch statement into a chain of if-then-else statements.
- The more efficient way is using a jump address table.
- The program needs only to index into the table and then jump to the appropriate sequence (address).

Case/Switch Statement

- Compiling a C code:

```
switch(k){
        case 0: f = i+j; break;
        case 1: f = g+h; break;
        case 2: f = g-h; break;
        case 3: f = i-j; break;
}
```

- Assume that register $t4 holds the base address of jumptable.



Case/Switch Statement

- Assembly code:

```
        sll $t1,$t1,2       # k = 4 * k (address problem)
        add $t1,$t1,$t4     # t1 = base address of jump table + 4*k
        jr $t1              # jump to addr pointed by t1
L0:     add $s0,$s3,$s4     # f = i+j
        j Exit
L1:     add $s0, $s1, $s2   # f = g+h
        j Exit
L2:     sub $s0, $s1, $s2   # f = g-h
        j Exit
L3:     sub $s0, $s3, $s4   # f = i-j
Exit:   <next instruction>
```

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load half	lh $s1,100($s2)	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	sh $s1,100($s2)	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte	lb $s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immed.	lui $s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Uncondi-tional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see Section 2.9)
jr	R	0	31	0	0	0	8	jr $ra
jal	J	3	2500					jal 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

Functions

- For the execution of a function (procedure),
  1. Place parameters, function can access them
  2. Determine the storage for function
  3. Perform the desired task.
  4. Place the result value where the caller can access it.
  5. Return control to the point of origin, since a function can be called from several points in a program.

Functions

- MIPS have specific registers to execute functions:
  - **$a0-$a3** four argument registers in which to pass parameters
  - **$v0-$v1** two value registers in which to return values
  - **$ra** one return address register to return to the point of origin

- MIPS has a special instruction (jump and link) just for functions

```
jal ProcedureAddress # jump the address and save the
                     # following address to the $ra
jr $ra               # jump back
```

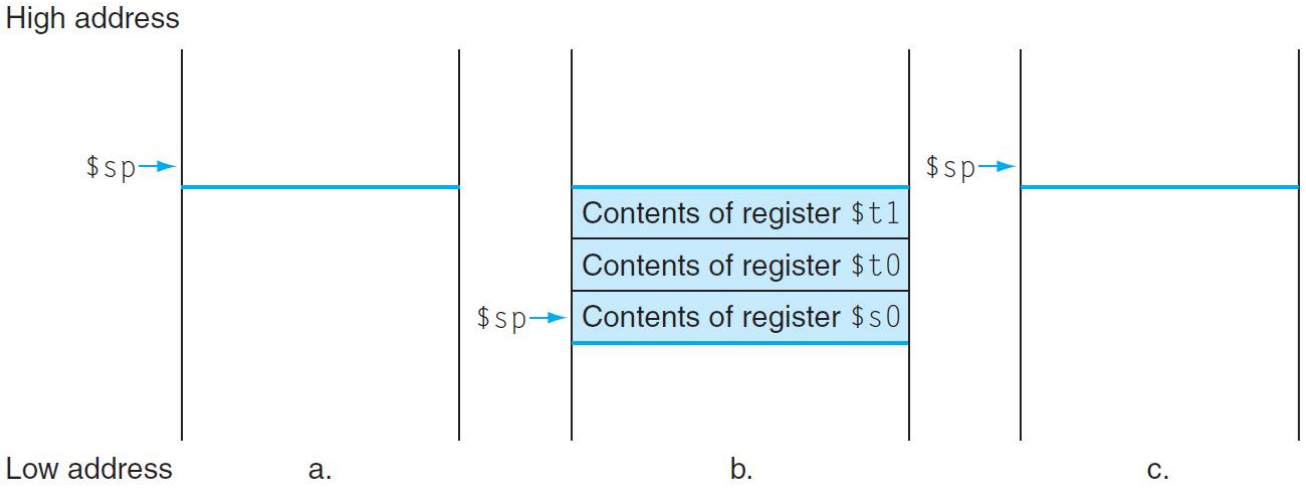
Functions

- When a function is called:
  1. The calling program (caller) puts the parameter values in $a0–$a3 and uses jal X to jump to function X.
  2. The function then performs the calculations, places the results in $v0–$v1,
  3. and returns control to the previous address using jr $ra.

- We must recover our previous values after our function is complete.
- Any registers needed by the caller must be restored to the values that they contained before the function was invoked.
- We need to take register values to memory before the function is called.

Stack Pointer

- The ideal data structure for this process is a stack.
- A stack needs a pointer (stack pointer) to the most recently allocated address in the stack.
- Stack pointer shows where the next function should place the registers or where old register values are found.



Stack Pointer

- The stack pointer ($sp) is a special register in MIPS,
- C code:

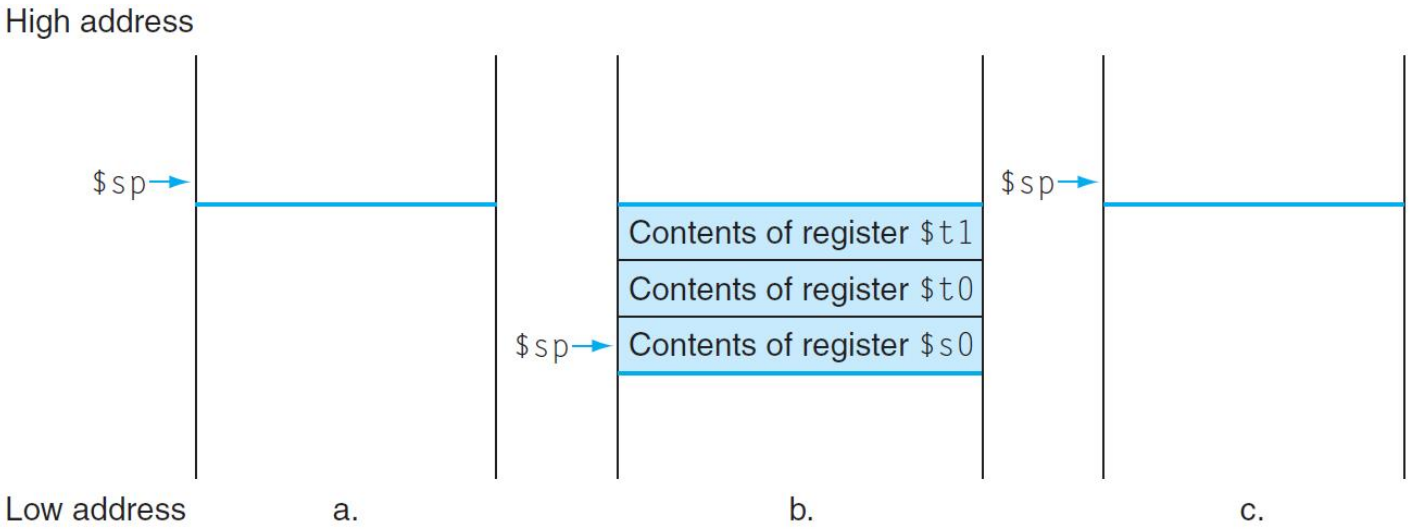
```
int leaf_example (int g, int h, int i, int j)
{
      int f;
      f = (g + h) - (i + j);
      return f;
}
```

Stack Pointer

- The parameter variables g, h, i, and j correspond to the argument registers $a0, $a1, $a2, and $a3, and f corresponds to $s0.

```
addi $sp,$sp,-12 # adjust stack to make room for 3 items
sw $t1, 8($sp)   # save register $t1 for use afterwards
sw $t0, 4($sp)   # save register $t0 for use afterwards
sw $s0, 0($sp)   # save register $s0 for use afterwards

add $t0,$a0,$a1   # register $t0 contains g + h
add $t1,$a2,$a3   # register $t1 contains i + j
sub $s0,$t0,$t1   # f = $t0 - $t1, which is (g + h)-(i + j)
```

```
add $v0,$s0,$zero      # returns f ($v0 = $s0 + 0)
lw $s0, 0($sp)         # restore register $s0 for caller
lw $t0, 4($sp)         # restore register $t0 for caller
lw $t1, 8($sp)         # restore register $t1 for caller
addi $sp,$sp,12        # adjust stack to delete 3 items
jr $ra                 # jump back to calling routine
```



Functions

- In the previous example temporary registers are used and assumed that their old values must be saved and restored.
- To avoid saving and restoring a register whose value is never used, MIPS separates 18 of the registers into two groups:
  - **$t0–$t9:** 10 temporary registers that does not have to be preserved
  - **$s0–$s7:** 8 saved registers that must be preserved.
- In the previous example registers $t0 and $t1 do not need to be preserved but still $s0 must be saved and restored.

Name	Register number	Usage	Preserved on call?
$zero	0	the constant value 0	n.a.
$v0–$v1	2–3	values for results and expression evaluation	no
$a0–$a3	4–7	arguments	no
$t0–$t7	8–15	temporaries	no
$s0–$s7	16–23	saved	yes
$t8–$t9	24–25	more temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp	Argument registers: $a0–$a3
Return address register: $ra	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer

In a function call

Nested Functions

- Nested Functions means functions invoking other functions.
- Suppose that the main program calls function A with an argument by placing it into register $a0.
- Then suppose that function A calls function B with an different argument also placed in $a0.
- Since A hasn't finished its task yet, there is a conflict over the use of register $a0. Similarly, there is a conflict over the return address in register $ra.
- The solution is saving and restoring **all of the registers**: argument registers ($a0–$a3), temporary registers ($t0–$t9) and return address register $ra and any saved registers ($s0– $s7).

Nested Functions

- C code:

```
int fact (int n)
{
    if (n < 1)
        return (1);
    else
        return (n * fact(n-1));
}
```

Nested Functions

- Assembly code:

```
fact: addi $sp,$sp,-8    # adjust stack for 2 items
      sw   $ra, 4($sp)   # save the return address
      sw   $a0, 0($sp)   # save the argument n
      slti $t0,$a0,1     # compare for n < 1
      beq  $t0,$zero,L1  # if n >= 1, go to L1

      addi $v0,$zero,1   # return 1
      addi $sp,$sp,8     # pop 2 items off stack
      jr   $ra           # return to after jal
```

Nested Functions

```
# Since $a0 and $ra don't change when n is less than 1
# we skip those instructions.
L1: addi $a0,$a0,-1   # n >= 1: argument gets (n - 1)
    jal fact          # call fact with (n - 1)

    lw $a0, 0($sp)    # return from jal:restore argument n
    lw $ra, 4($sp)    # restore the return address
    addi $sp,$sp,8    # adjust stack pointer to pop 2 items
   # assume a multiply instruction is available
    mul $v0,$a0,$v0   # return n * fact (n - 1)
    jr $ra            # return to the caller
```

Nested Functions

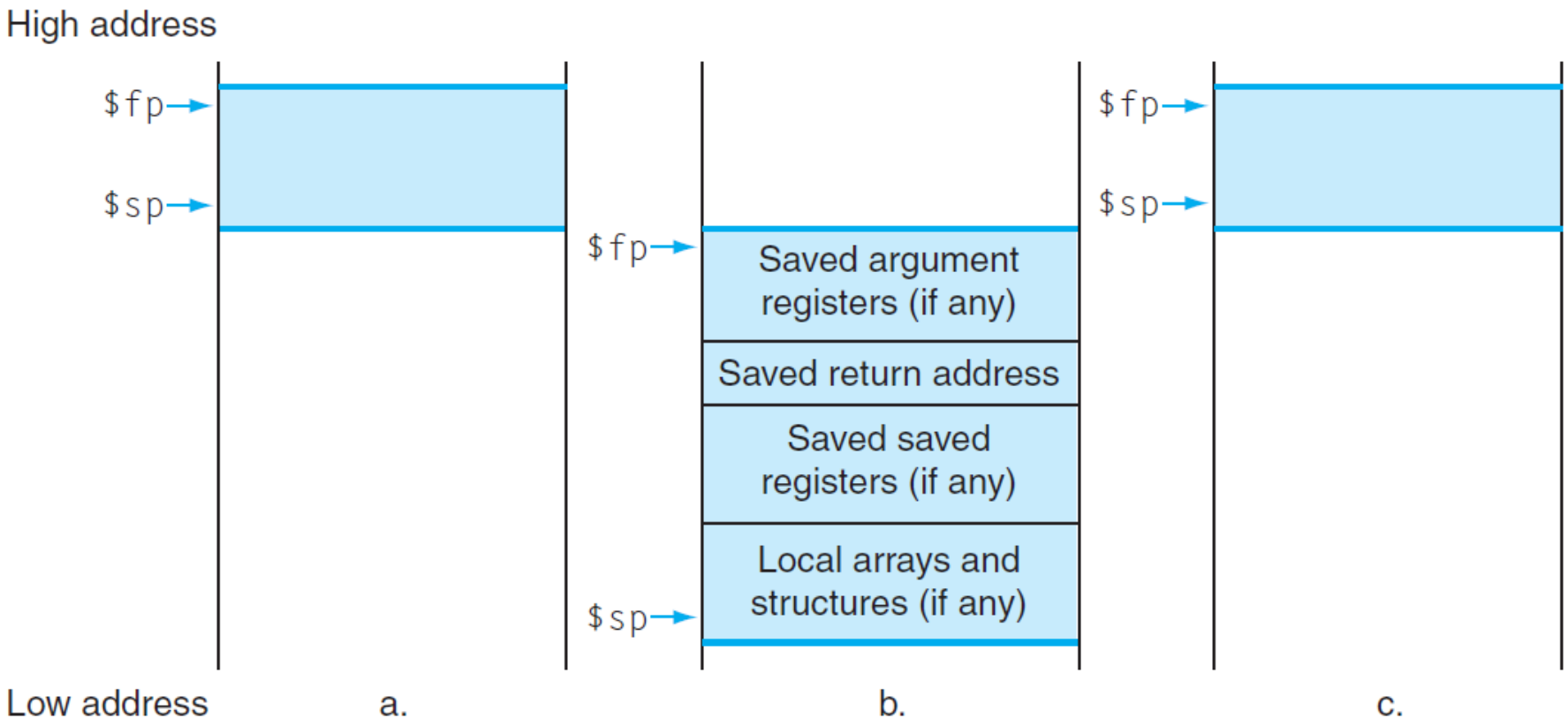
- C has two storage classes: automatic and static. Automatic variables are local to a procedure and are discarded when the function exits.
- Static variables exist across exits from and entries to functions. C variables declared outside and variables declared using the keyword static are considered static.
- To simplify access to static data, MIPS reserves another register, called the **global pointer**, or **$gp**.

Stack & Frame Pointers

- Stack is also used to store variables that are local to the function that do not fit in registers, such as arrays, addressable by stack pointer.
- MIPS also uses a **frame pointer ($fp)** to point to the first word of the stack.
- If there are more than four parameters, the extra parameters is placed on the stack just above the frame pointer (in some compilers)
- The function expects the first four parameters to be in registers $a0 through $a3 and the rest in memory, addressable via the frame pointer.

- Another useful feature of frame pointer is that it’s easier for programmers to reference the beginning of the stack via the stable frame pointer.

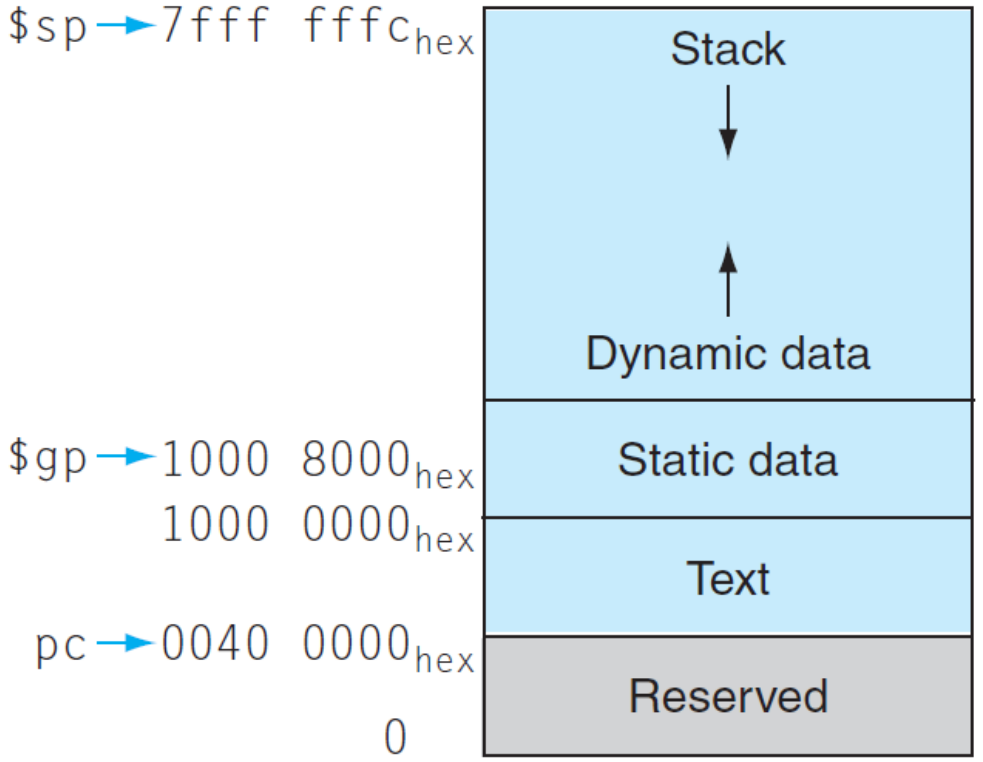
Frame Pointer



a) Before function call
b) During function call
c) After function call

Memory Space

- The first part of the memory is reserved, followed by the home of the MIPS machine code, traditionally called the text segment.
- Static data segment is the place for constants and other static variables.
- Dynamic data tends to grow and shrink during their lifetimes.
- This allocation allows the stack and dynamic data to grow toward each other.



Strings

- Most computers use 8-bit bytes to represent characters as ASCII.
- Unicode is an another universal encoding of the alphabets. It uses 16 bits to represent a character by default.

- MIPS provides instructions to move bytes.
  - `lb $t0,0($sp)    # Read byte from source`
  - `sb $t0,0($gp)    # Write byte to destination`
- Load byte (lb) loads a byte from memory, placing it in the rightmost 8 bits of a register.
- Store byte (sb) takes a byte from the rightmost 8 bits of a register and writes it to memory.

Strings

- Strings have a variable number of characters.
- There are three choices for representing a string:
  1. The first position of the string is reserved to give the length of a string,
  2. An accompanying variable has the length of the string,
  3. The last position of a string is indicated by a character used to mark the end of a string,

- C language uses the third choice, terminating a string with a byte whose value is 0 (named null in ASCII).

Strings

- C code:

```
void strcpy (char x[], char y[])
{
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0' )    // copy & test byte
          i += 1;
}
```

Strings

- Assembly

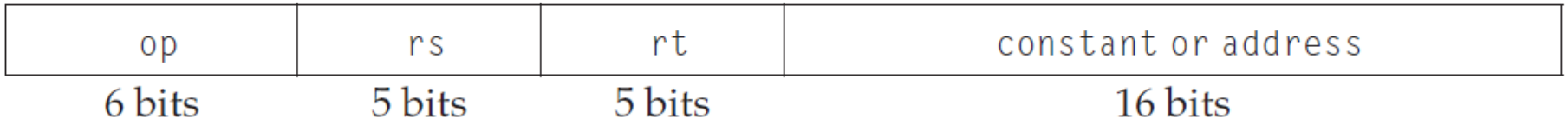
```
strcpy:
        addi $sp,$sp,-4          # adjust stack for 1 more item
        sw   $s0, 0($sp)         # save $s0 (i)
        add  $s0,$zero,$zero     # i = 0 + 0 (to initialize)

L1:     add $t1,$s0,$a1          # address of y[i] in $t1
        lb  $t2, 0($t1)          # $t2 = y[i]
        add $t3,$s0,$a0          # address of x[i] in $t3
        sb  $t2, 0($t3)          # x[i] = y[i]
        beq $t2,$zero,L2         # if y[i] == 0, go to L2
```

Strings

```
# If not, we increment i and loop back:
    addi $s0, $s0,1     # i = i + 1
    j    L1             # go to L1

L2: lw $s0, 0($sp)       # y[i] == 0: end of string;
                         # restore old $s0
    addi $sp,$sp,4       # pop 1 word off stack
    jr $ra               # return
```

32-bit Constants

- **32-Bit Immediate Operands:**



- The MIPS includes an instruction load upper immediate (lui) to set the **upper 16 bits** of a constant in a register.

32-bit Constants

The machine language version of `lui $t0, 255` `# $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing `lui $t0, 255:`

0000 0000 1111 1111	0000 0000 0000 0000

- The instruction **lui** transfers the 16-bit immediate constant
- Field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.

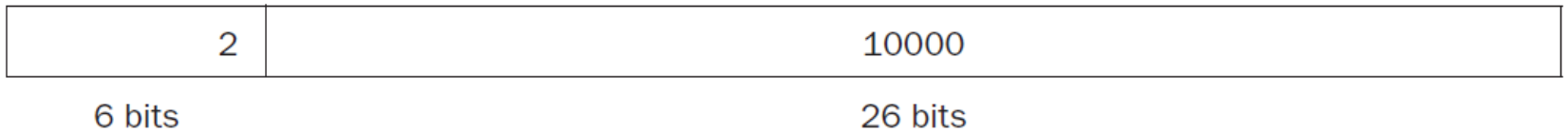
32-bit Constants

```
# 0000 0000 0011 1101 0000 1001 0000 0000
  lui $s0, 61     # 61 = 0000 0000 0011 1101 binary

# The value of register $s0 afterward is
# 0000 0000 0011 1101 0000 0000 0000 0000

  ori $s0, $s0, 2304    # 2304 = 0000 1001 0000 0000   (or immediate)

# The final value in register $s0 is the desired value:
# 0000 0000 0011 1101 0000 1001 0000 0000
```

Addressing in Branches

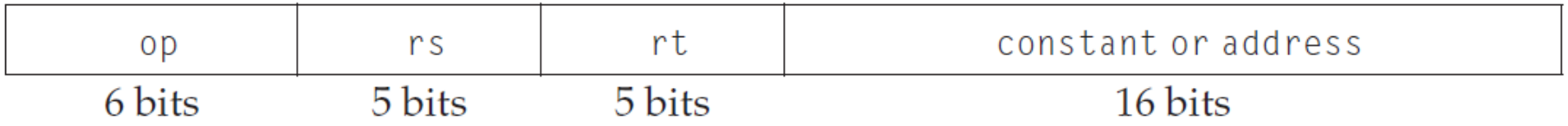
- For unconditional jumps, J-type format is used.


```
j 10000 # go to location 10000
```

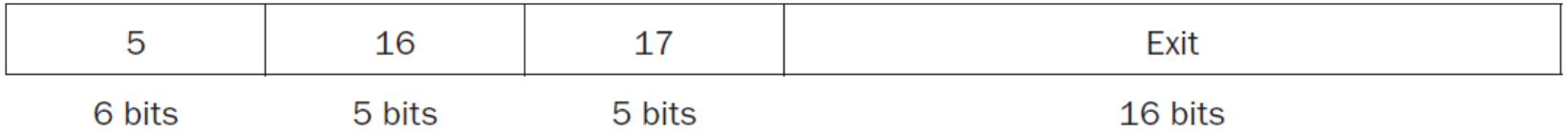


Addressing in Branches

- For conditional jumps, I-type format is used.




```
bne $s0,$s1,Exit    # go to Exit if $s0 != $s1
```



Addressing in Branches

- A branch instruction can be calculated with the following
  - Program counter = Current Address + Branch address

- PC allows the program to be as large as 2^32 addresses (words) = 16 GB.
- However branching can be done within ±2^16 words of the current instruction.
- This is called **PC-relative addressing.**
- MIPS instructions are 4 bytes long and addressing refer to the words 32-bit not bytes.

Addressing in Branches

```
Loop: sll  $t1,$s3,2
      add  $t1,$t1,$s6
      lw   $t0,0($t1)
      bne  $t0,$s5, Exit
      addi $s3,$s3,1
      j Loop
Exit:
```

Instructions and Their Addresses

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

Addressing in Branches

- The loader and linker must be careful to avoid placing a program across an address boundary of 256 MB (2^16 words).
- Otherwise a jump can be replaced by a jump register instruction (jr) and the register must be loaded a full 32-bit address. (like the 32-bit constants)

The machine language version of `lui $t0, 255` `# $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing `lui $t0, 255`:

0000 0000 1111 1111	0000 0000 0000 0000

Addressing in Branches

- Another method for branching far away is using unconditional branch instructions (2^26 words).

```
beq $s0,$s1, L1
```

```
bne $s0,$s1, L2
j L1
L2:
```

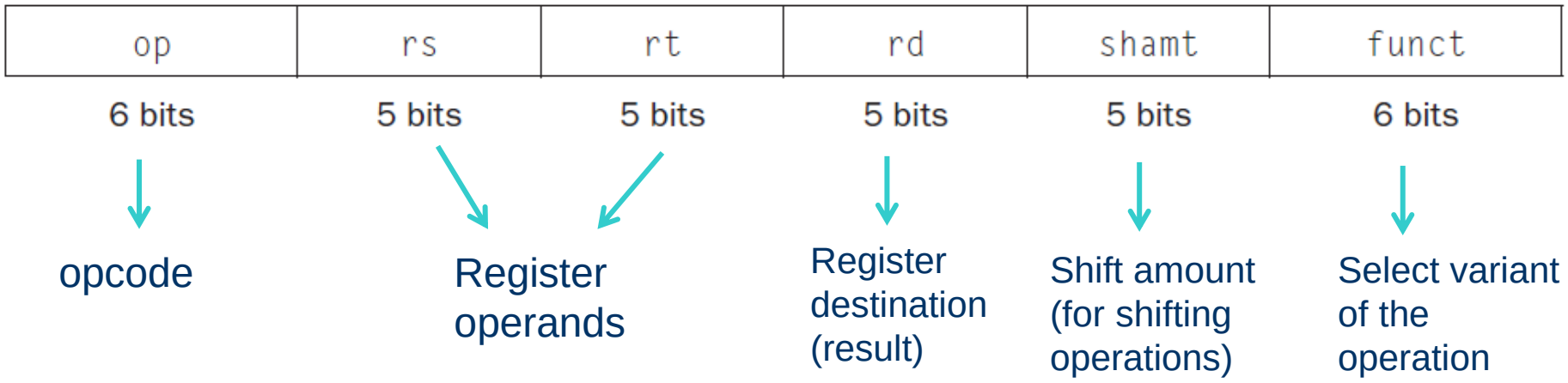
offers a much greater branching distance

Addressing Modes

- Multiple forms of addressing are called **addressing modes**. The MIPS addressing modes are the following:
  1. **Register addressing**, where the operand is a register
  2. **Immediate addressing**, where the operand is a constant within the instruction itself
  3. **Base** or **displacement addressing**, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
  4. **PC-relative addressing**, where the address is the sum of the PC and a constant in the instruction
  5. **Pseudodirect addressing**, where the jump address is the 26 bits of the instruction concatenated with PC

Addressing Modes

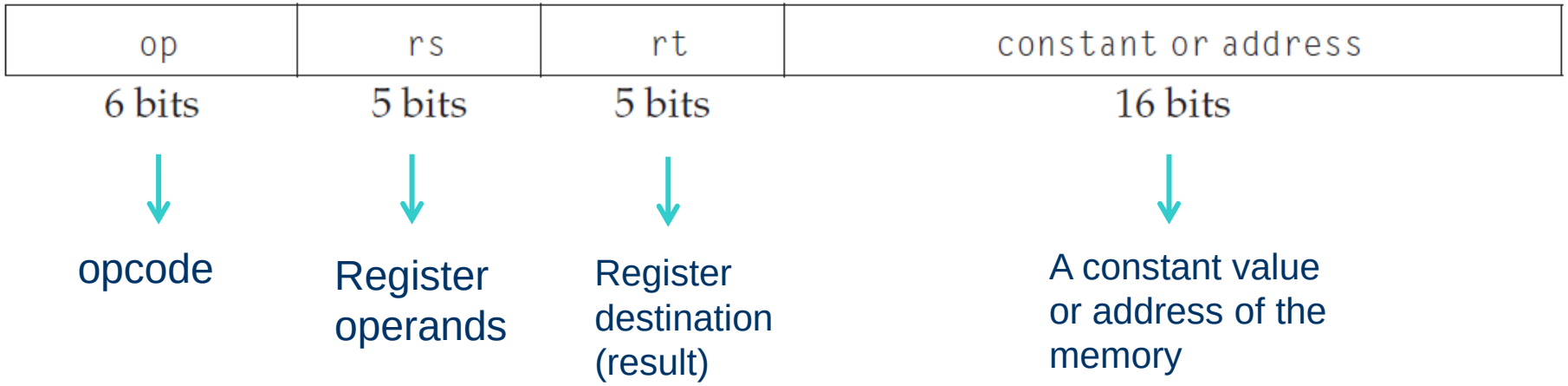
1. **Register Addressing:**



- R-type instructions

Addressing Modes

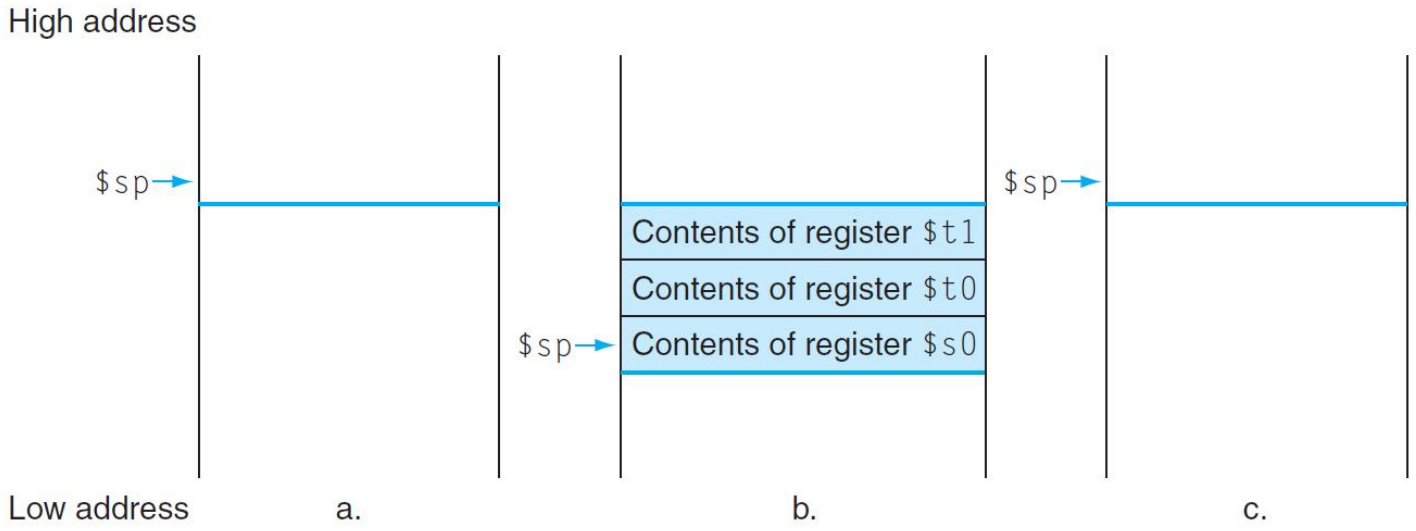
2. Immediate Addressing:



- I-type instructions

Addressing Modes

3. Base or Displacement Addresing:



- Using stacks for functions

Addressing Modes

4. **PC-relative addressing**, where the address is the sum of the PC and a constant in the instruction

Conditional branch	branch on equal	`beq $s1,$s2,L`	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	`bne $s1,$s2,L`	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; used with `beq`, `bne`
	set on less than immediate	`slt $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than immediate; used with `beq`, `bne`
Unconditional jump	jump	`j L`	go to L	Jump to target address
	jump register	`jr $ra`	go to $ra	For procedure return
	jump and link	`jal L`	$ra = PC + 4; go to L	For procedure call

- Conditional and unconditional branching

Addressing Modes

5. **Pseudodirect addressing**, where the jump address is the 26 bits of the instruction concatenated with PC

```
beq $s0,$s1, L1
```

```
bne $s0,$s1, L2
j L1
L2:
```

offers a much greater branching distance

- Branching far away (more than 2^16)

Decoding Machine Language

- What is the assembly language statement corresponding to this machine instruction?

  00af8020hex

0000 0000 1010 1111 1000 0000 0010 0000 bin

We can look at the op field to determine the operation.

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

- when bits 31–29 are 000 and bits 28–26 are 000, it is an R-format instruction.

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	sltu				
6(110)								
7(111)								

- In this case, bits 5–3 are 100 and bits 2–0 are 000, which means this binary pattern represents an add instruction.

Decoding Machine Language

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

- These numbers represent registers $a1, $t7, and $s0.
- Now we can show the assembly instruction:

```
add $s0,$a1,$t7
```

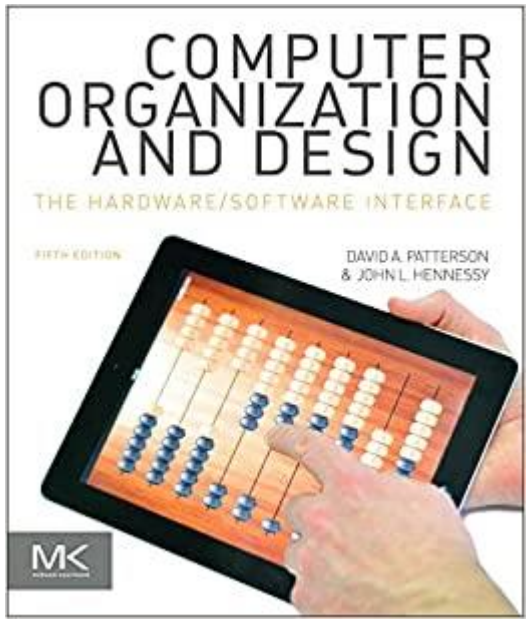
Name	Register number
$zero	0
$v0–$v1	2–3
$a0–$a3	4–7
$t0–$t7	8–15
$s0–$s7	16–23
$t8–$t9	24–25
$gp	28
$sp	29
$fp	30
$ra	31

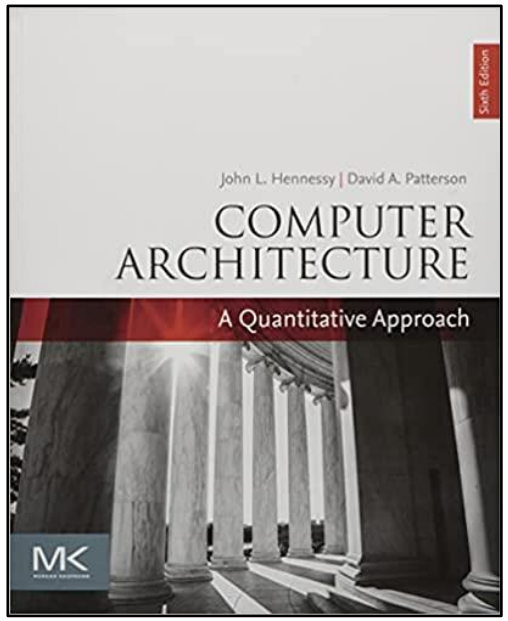
References

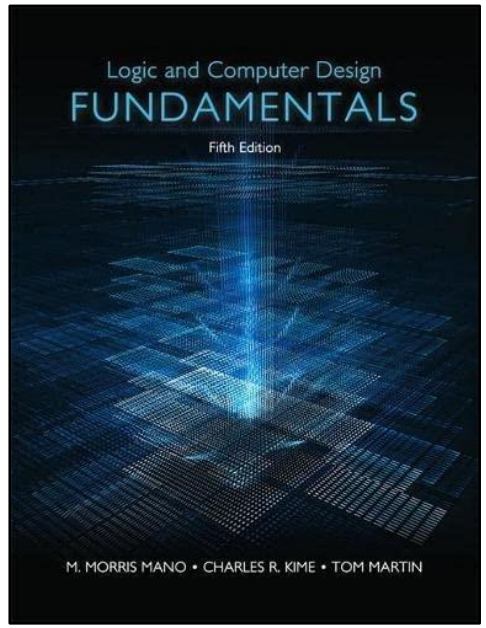
- EHB326E - Introduction to Embedded Systems, Müştak Erhan Yalçın https://web.itu.edu.tr/yalcinmust/ehb326.html
- Xilinx, Introduction to FPGA Design with Vivado High-Level Synthesis, January 22, 2019.
- D. A. Patterson, *Computer organization and Design*. San Francisco: Elsevier Science & Technology.
- M. M. Mano, and C. R. Kime,, *Logic and computer design fundamentals*. Boston: Pearson.
- Computer Architecture, Princeton University
- https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

Further&Advanced Information

- D. A. Patterson, *Computer organization and Design*. Morgan Kaufmann.
- J. L. Hennessy and D. A. Patterson, *Computer architecture A quantitative approach.* Morgan Kaufmann.
- M. M. Mano, and C. R. Kime,, *Logic and computer design fundamentals*. Pearson.







Further&Advanced Information

- First 3 weeks of the course are recommended.
- It is free of charge.

- “Computer Architecture,” *Coursera*. [Online]. Available: https://www.coursera.org/learn/comparch. [Accessed: 03-Apr-2021].