Computer Organization and Operating System

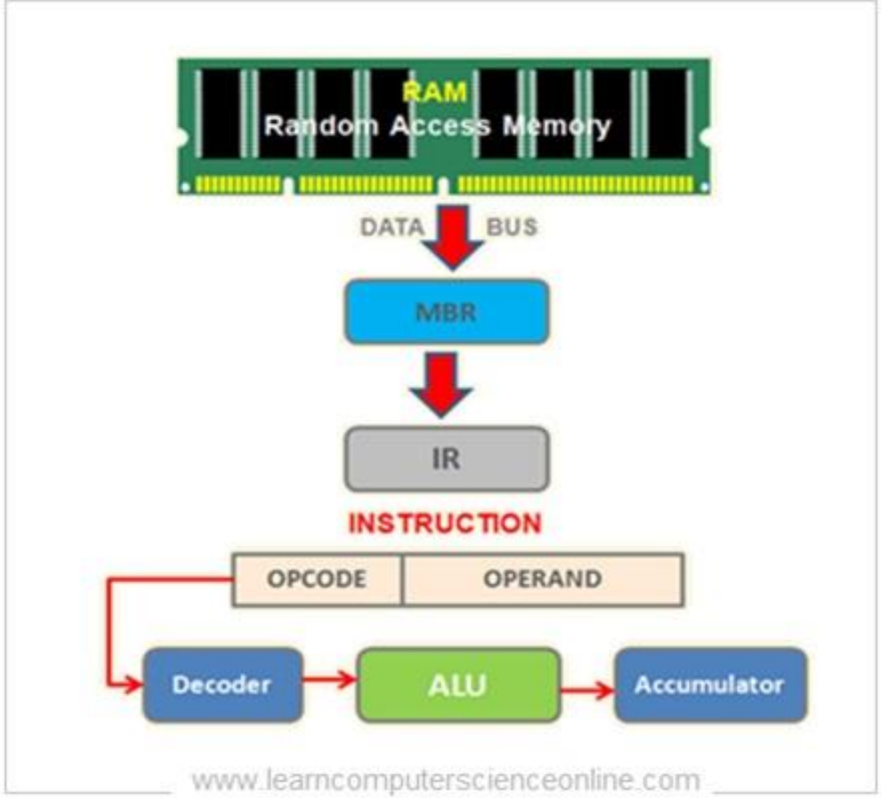
Instruction Execution

Akharin Khunkitti

KMITL

Topic

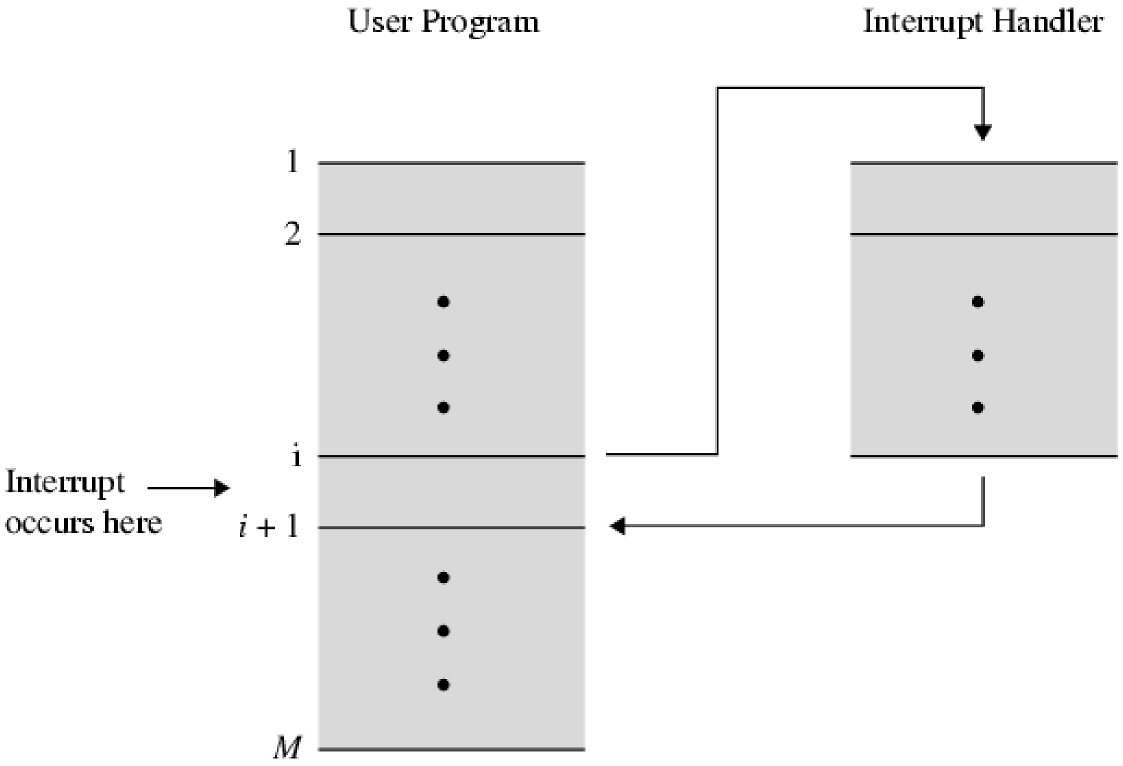
- Interruption
- Instruction Cycle
  - Fetch, Execution, Interrupt
- Instruction Set Architecture (ISA)
  - Addressing Mode
  - Register Model
  - Instruction Format
  - Byte-Order
- Computer Architecture - CISC and RISC
- Conclusion



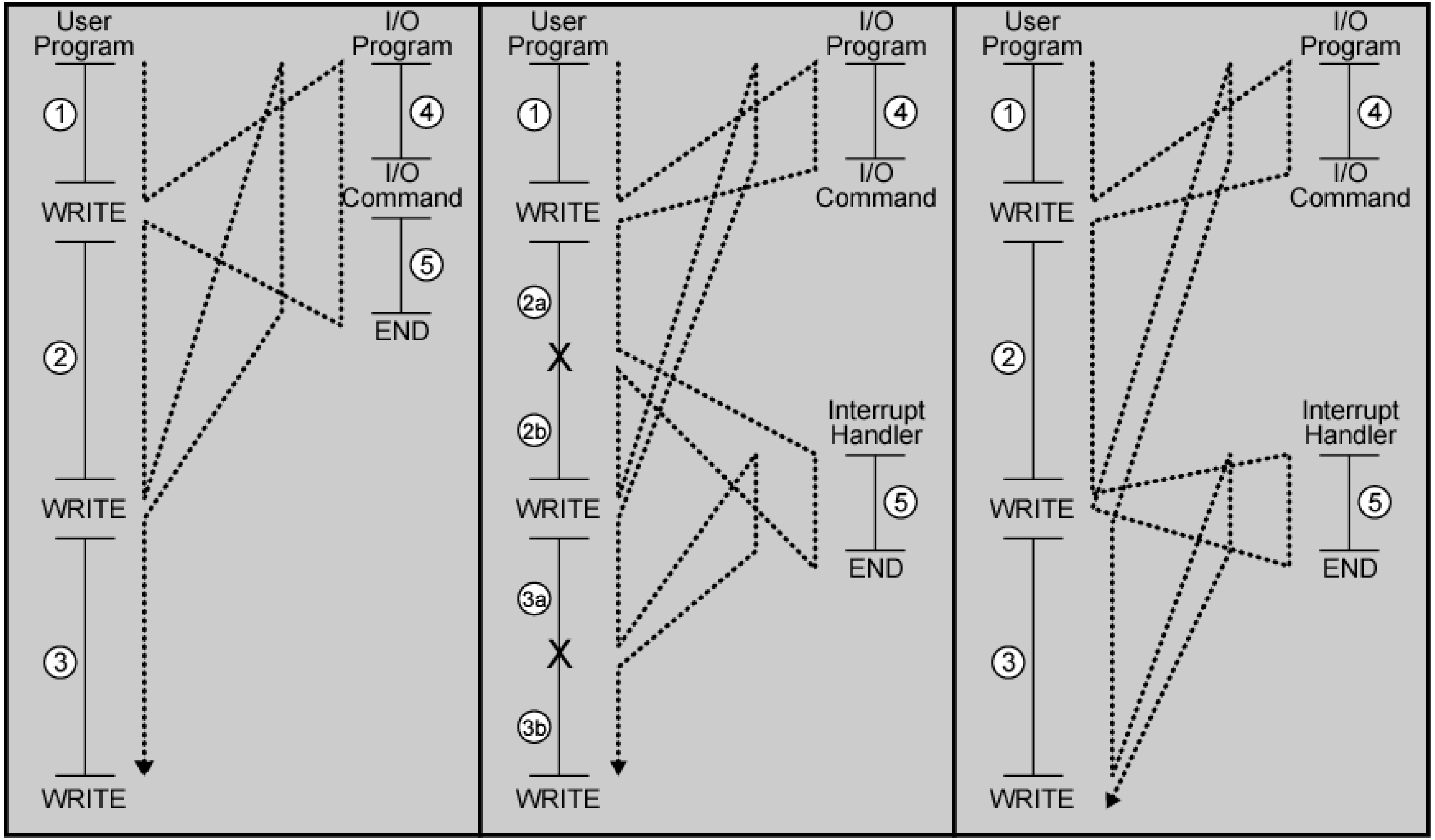
www.learncomputerscienceonline.com

Program Interruption

- Execute other programs, which have higher priority.
- Interrupts - Mechanism by which other modules (e.g. I/O) may interrupt normal sequence of processing
  - Program
    - e.g. overflow, division by zero
  - Timer
    - Generated by internal processor timer
    - Used in pre-emptive multi-tasking
  - I/O
    - from I/O controller
  - Hardware failure
    - e.g. memory parity error



Program Flow Control



(a) No interrupts

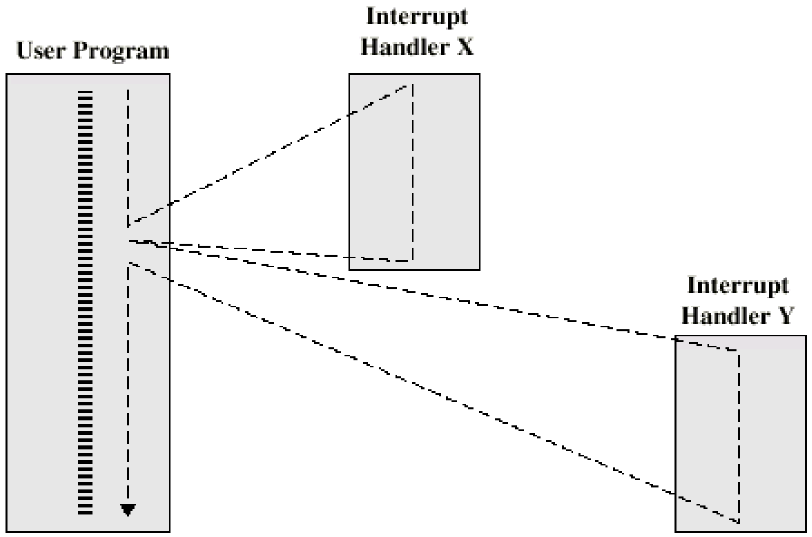
(b) Interrupts; short I/O wait

(c) Interrupts; long I/O wait

Multiple Interrupts

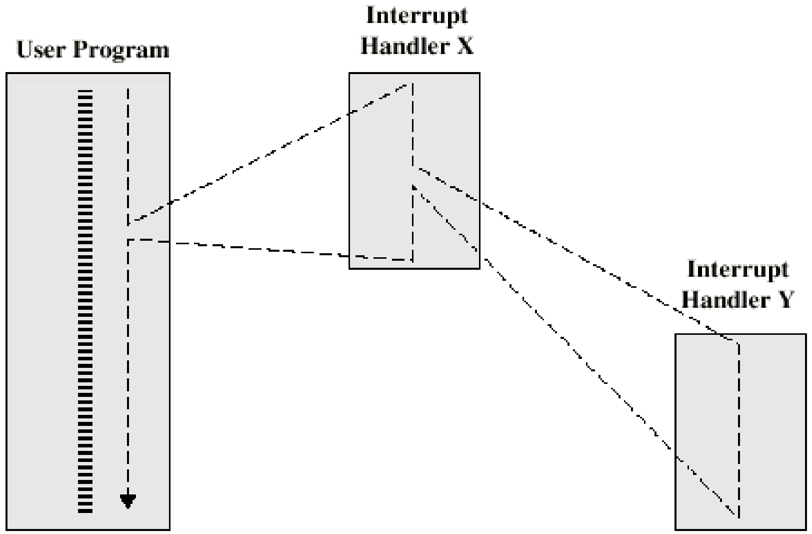
- Program flows can be interrupted multiple times
- Multiple Interrupts Types;
  - Sequential Interrupts
    - Interrupt after another interrupt completed

Sequential



  - Nested Interrupts
    - Interrupt during another interrupt executing
      - not completed yet

Nested

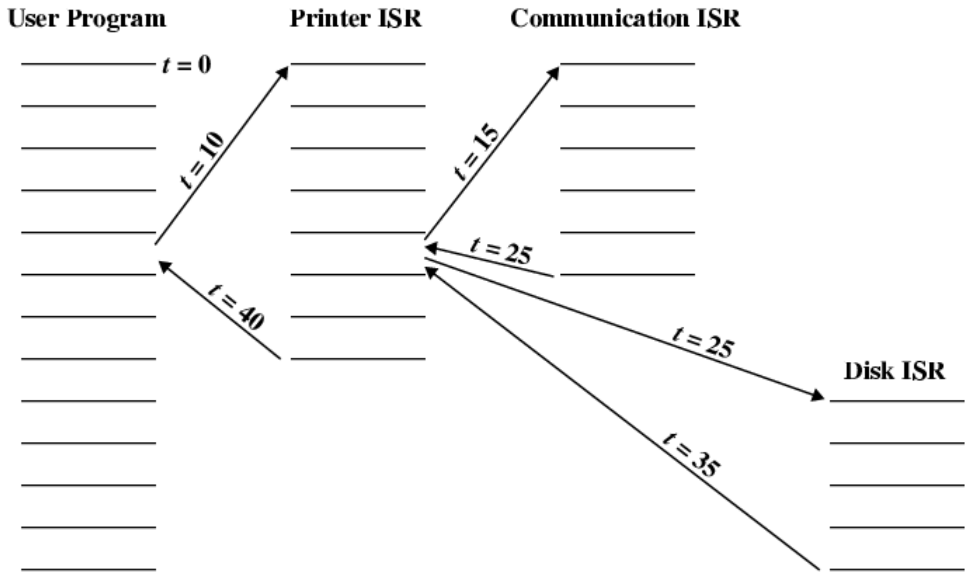


Multiple Interrupts

Best Practices for Multiple Interrupts

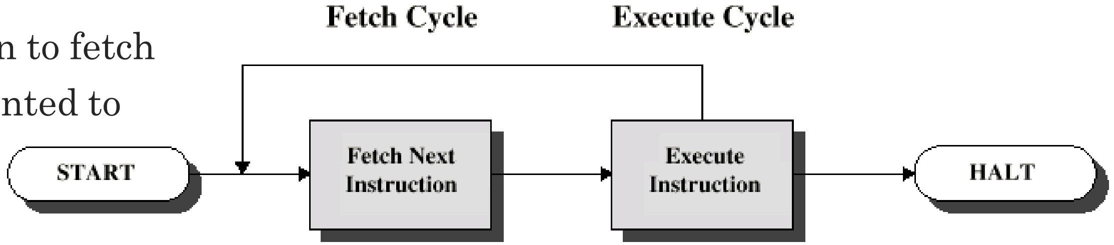
- Disable interrupts
  - Processor will ignore further interrupts while processing one interrupt
  - Interrupts remain pending and are checked after first interrupt has been processed
  - Interrupts handled in sequence as they occur
- Define priorities
  - Low priority interrupts can be interrupted by higher priority interrupts
  - When higher priority interrupt has been processed, processor returns to previous interrupt
    - Like sequential

Example: Time Sequence of Multiple Interrupts

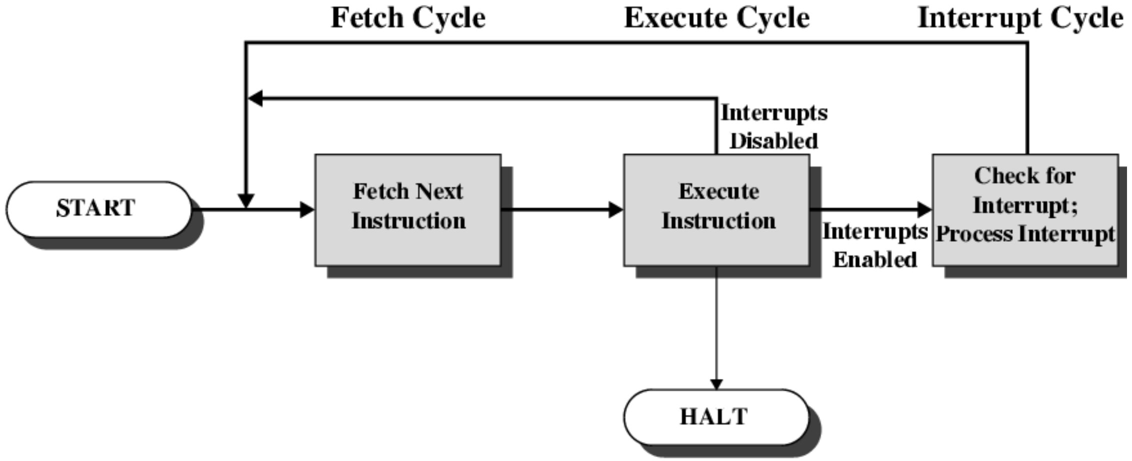


Instruction Cycle

- Basic Instruction Cycle (No Interrupt)
  - Fetch
    - Program Counter (PC) holds address of next instruction to fetch
    - Processor fetches instruction from memory location pointed to by PC
    - Increment PC
    - Unless told otherwise
    - Instruction loaded into Instruction Register (IR)
    - Processor interprets instruction and performs required actions
  - Execute
    - Operate by control signals, from Control Unit

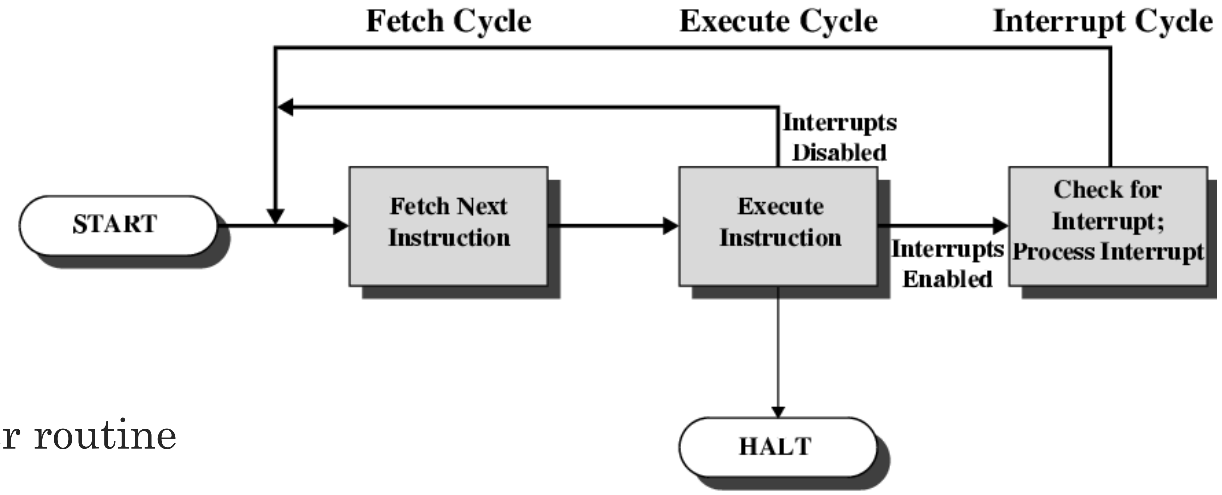


- Instruction Cycle with Interrupt
  - Fetch
  - Execute
  - Interrupt

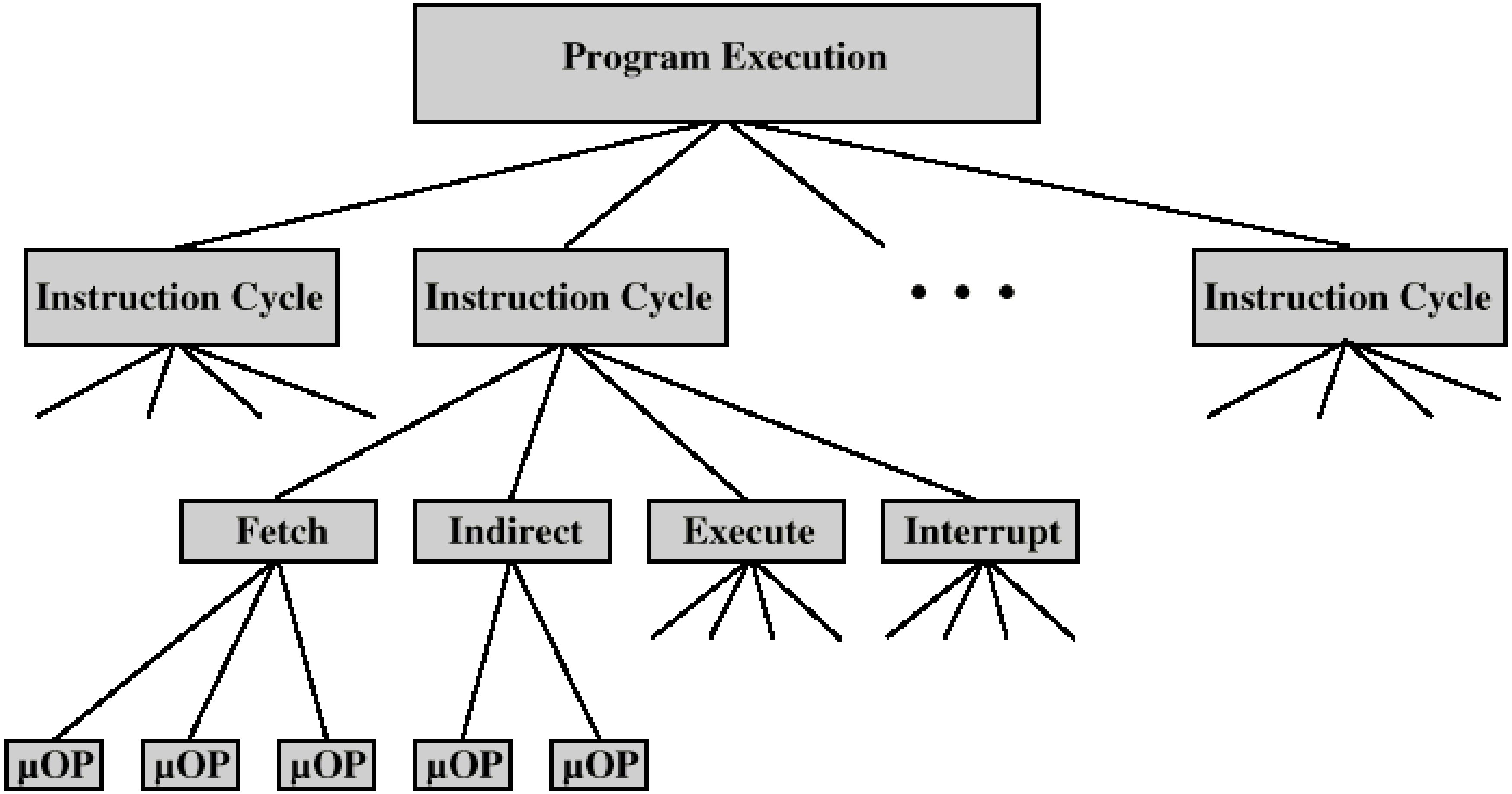


Interrupt Cycle

- Added to instruction cycle
- Processor checks for interrupt
  - Indicated by an interrupt signal
- If no interrupt, fetch next instruction
- If interrupt pending:
  - Suspend execution of current program
  - Save context
  - Set PC to start address of interrupt handler routine
  - Process interrupt
  - Restore context and continue interrupted program



Elements of Program Execution

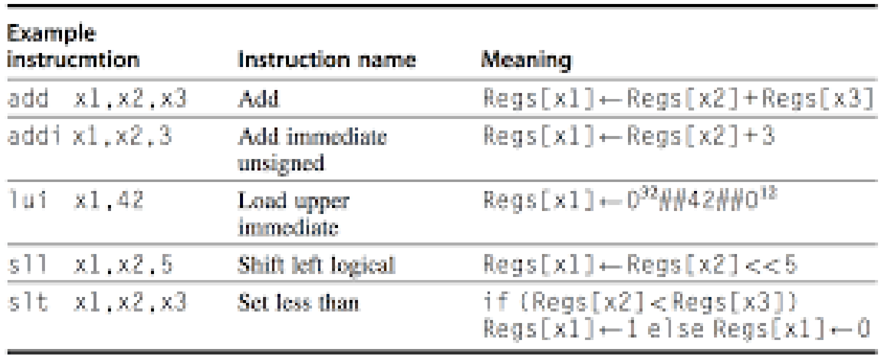


Instruction Set Architecture

What is an Instruction Set?

- Instructions
  - Codes to be interpreted or decoded by CPU

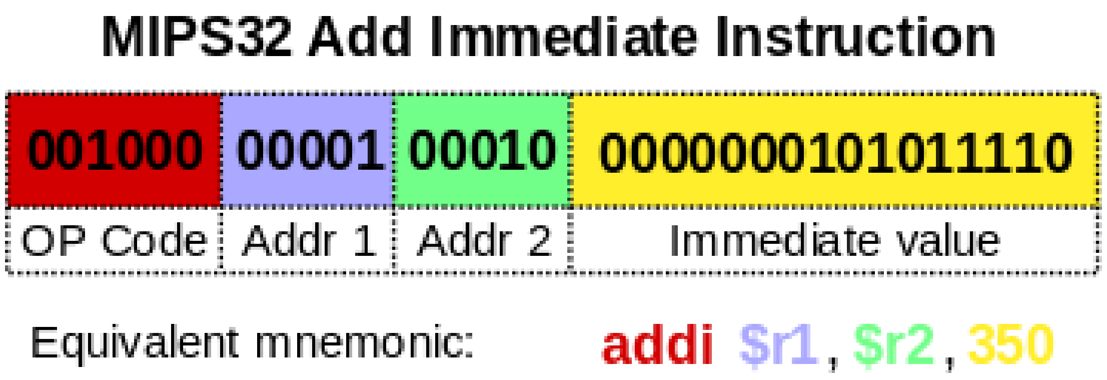
    => Operation execution
- Instruction Set
  - The complete collection of instructions that are understood by a CPU
  - Machine Code => Binary => for CPU operations
  - Usually represented by assembly codes => for human working

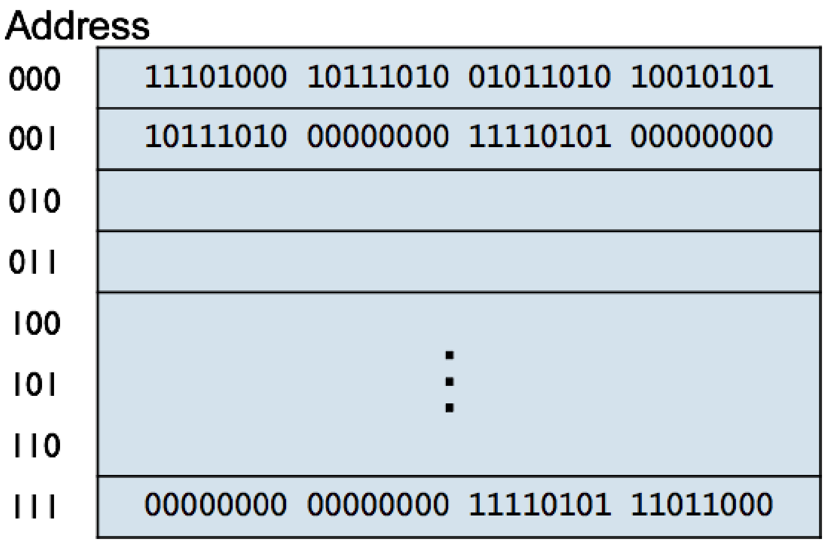


Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	`Regs[x1]←Regs[x2]+Regs[x3]`
`addi x1,x2,3`	Add immediate unsigned	`Regs[x1]←Regs[x2]+3`
`lui x1,42`	Load upper immediate	`Regs[x1]←`0^{32}`##42##`0^{12}
`sll x1,x2,5`	Shift left logical	`Regs[x1]←Regs[x2]<<5`

`slt x1,x2,x3`	Set less than	`if (Regs[x2]<Regs[x3])`
		`Regs[x1]←1 else Regs[x1]←0`

Figure A.26 The basic ALU instructions in RISC-V are available both with register-register operands and with one immediate operand. LUI uses the U-format that employs the rs1 field as part of the immediate, yielding a 20-bit immediate.





Address	
000	11101000 10111010 01011010 10010101
001	10111010 00000000 11110101 00000000
010	
011	
100	
101	⋮
110	
111	00000000 00000000 11110101 11011000

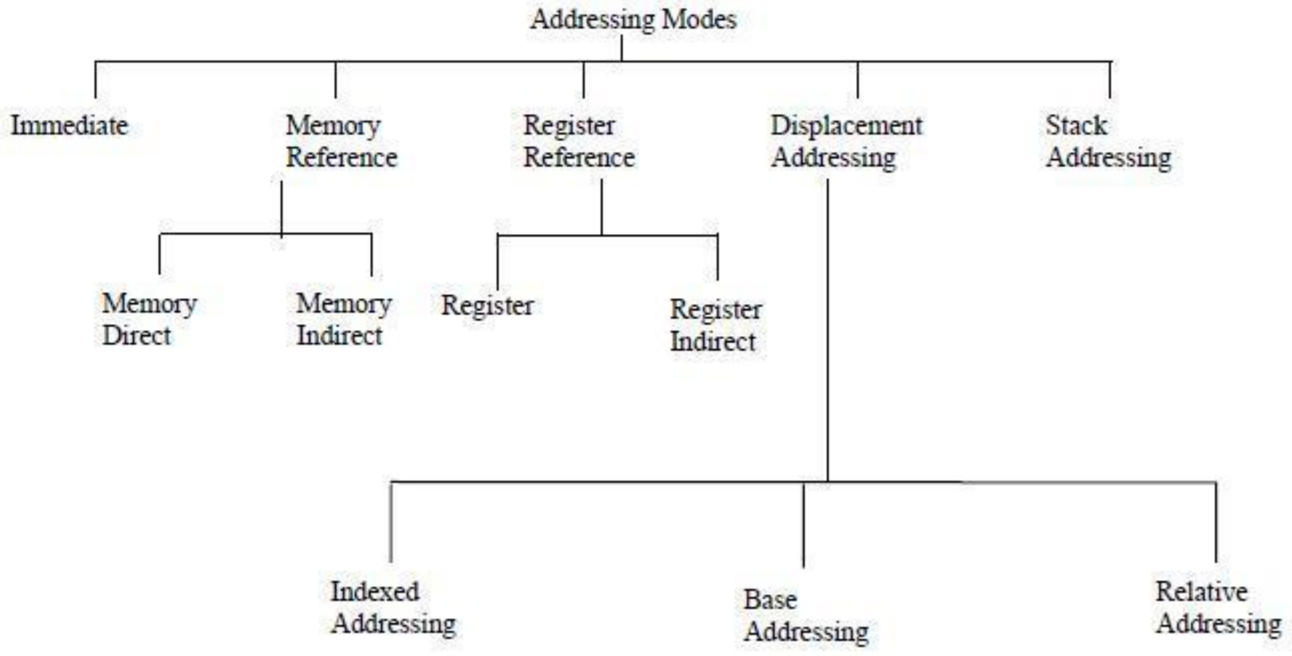
Addressing Modes

Fundamental Modes

- Immediate
- Direct
- Indirect

Alternative Modes

- Register
- Register Indirect
- Displacement (Indexed)
- Stack



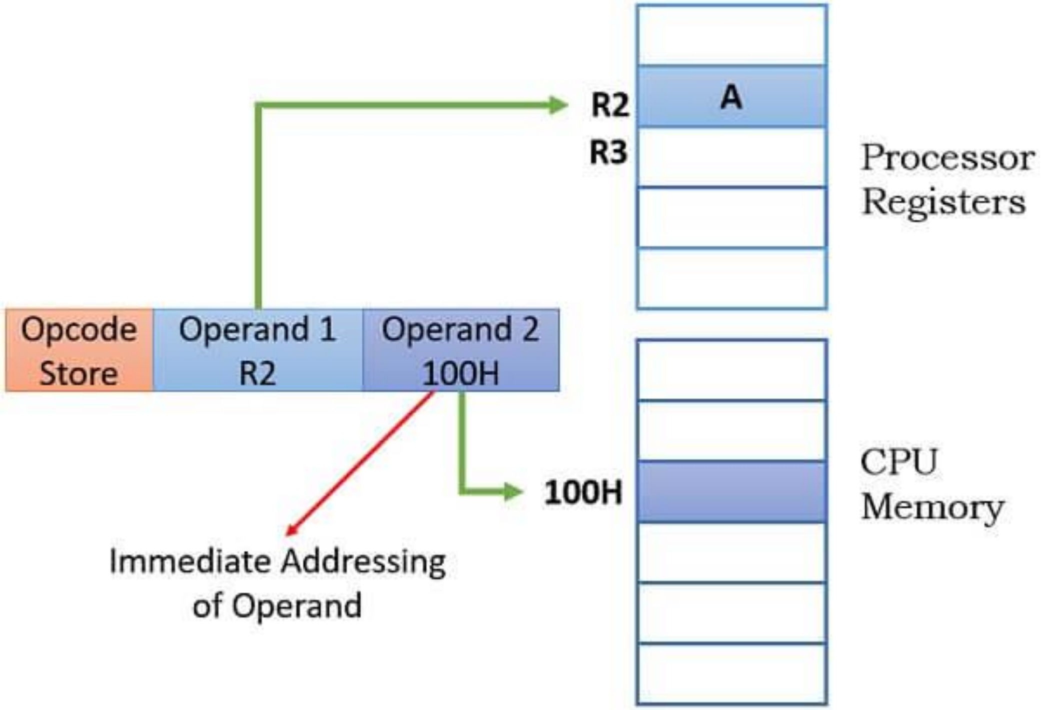
Common Addressing Modes

Immediate Addressing

- Operand is part of instruction
- Operand = address field
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

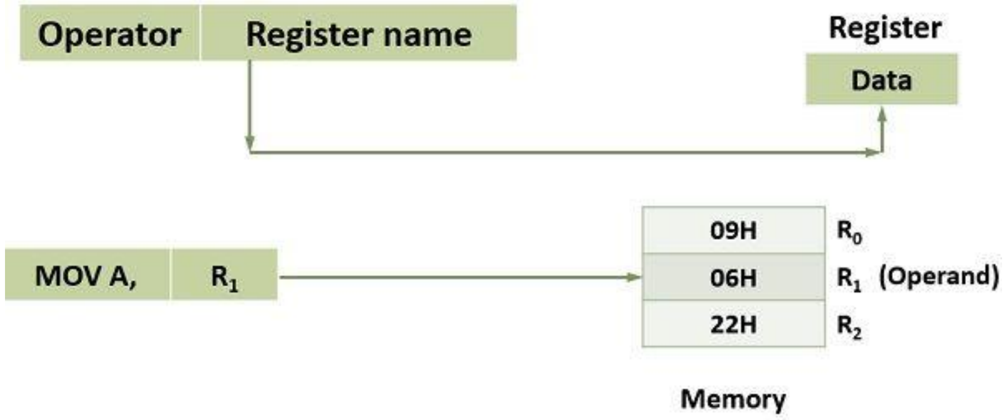
Instruction

Opcode	Operand

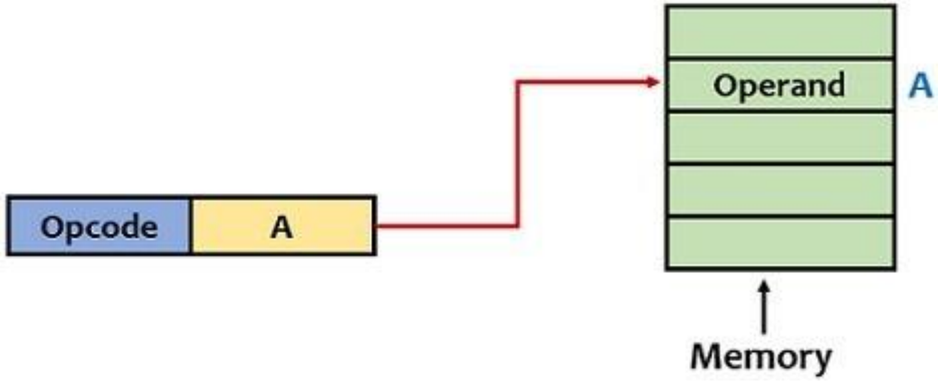


Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g. ADD A
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space

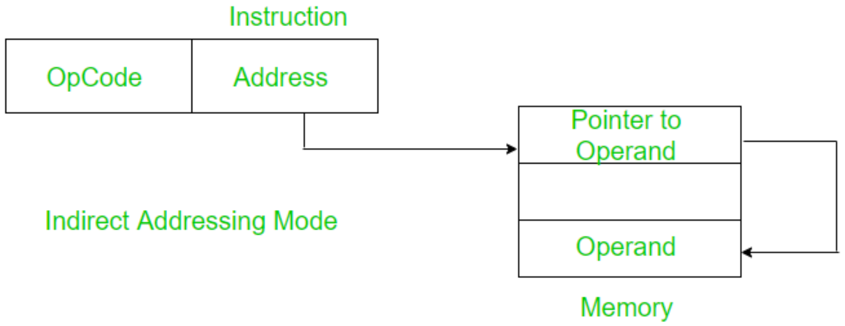


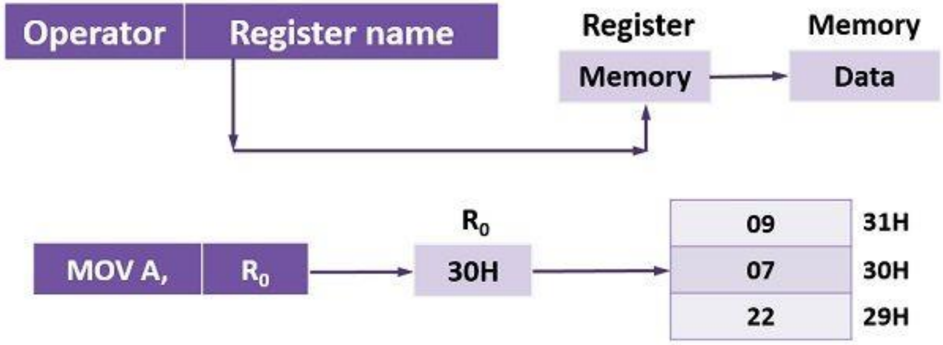
Direct Addressing Example



Indirect Addressing

- Memory cell pointed to by address field contains the address of (pointer to) the operand
- EA = (A)
  - Look in A, find address (A) and look there for operand
- e.g. ADD (A)
  - Add contents of cell pointed to by contents of A to accumulator
- Large address space
- 2^n where n = word length
- May be nested, multilevel, cascaded
  - e.g. EA = (((A)))
    - Draw the diagram yourself
- Multiple memory accesses to find operand
- Hence slower

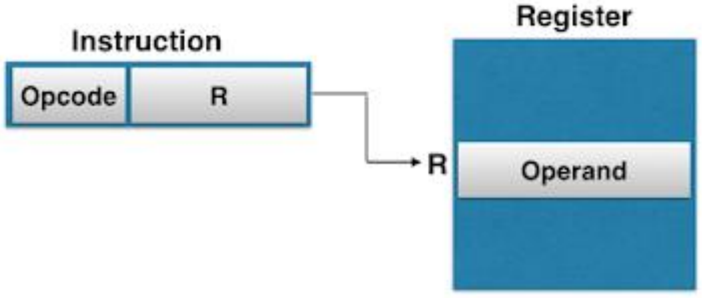


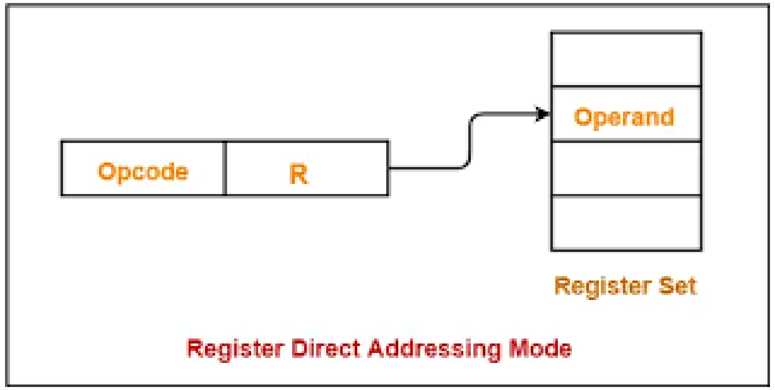


Indirect Addressing Example

Register Addressing

- Operand is held in register named in address filed
- EA = R
- Limited number of registers
- Very small address field needed
  - Shorter instructions
  - Faster instruction fetch
- No memory access
- Very fast execution
- Very limited address space
- Multiple registers helps performance
  - Requires good assembly programming or compiler writing
  - N.B. C programming
    - register int a;
- A kind of “Direct addressing”

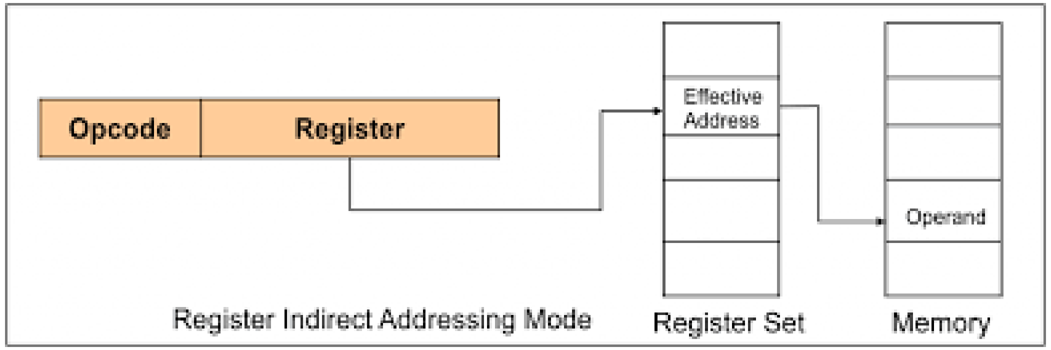


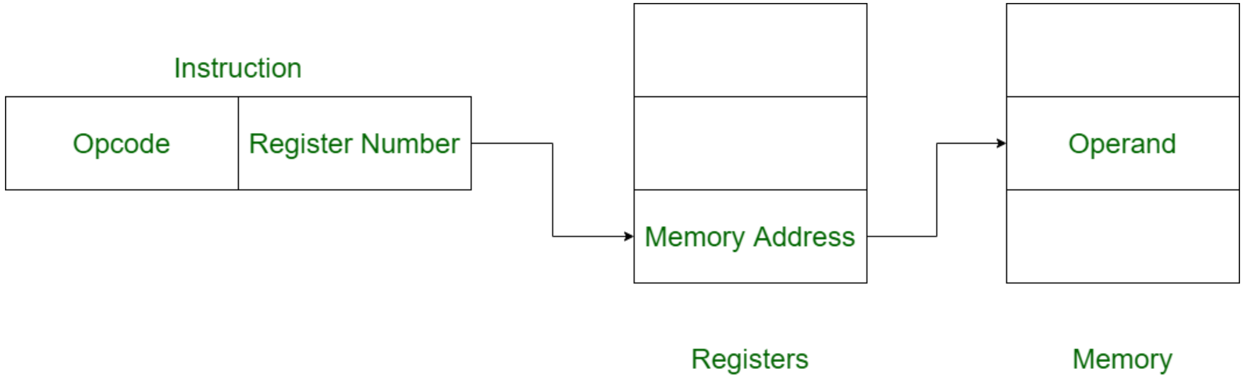


Register Direct Addressing Mode

Register Indirect Addressing

- A kind of “Indirect addressing”
- EA = (R)
- Operand is in memory cell pointed to by contents of register R
- Large address space (2^n)
- One fewer memory access than indirect addressing

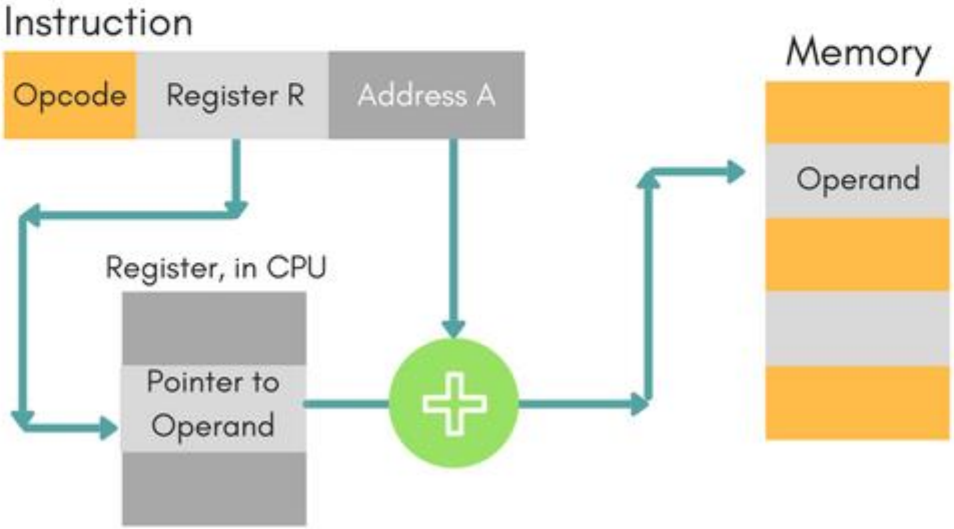


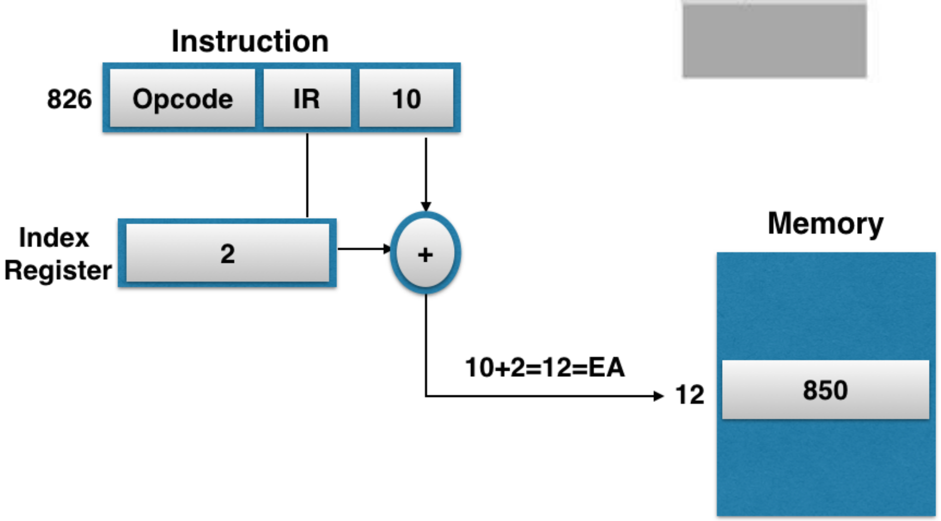


Register Indirect Mode

Displacement Addressing

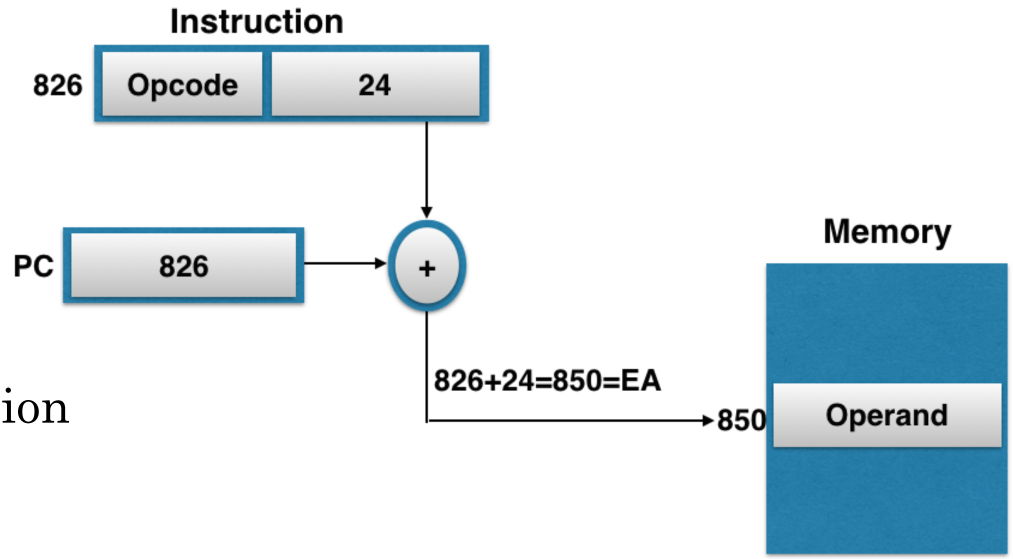
- EA = A + (R)
- Address field hold two values
  - A = base value
  - R = register that holds displacement
  - or vice versa

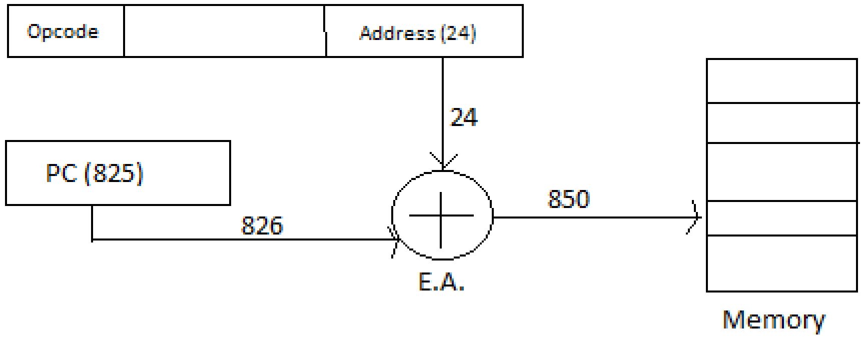




Relative Addressing

- A version of displacement addressing
- R = Program counter, PC
- EA = A + (PC)
- i.e. get operand from A cells from current location pointed to by PC
- c.f locality of reference & cache usage

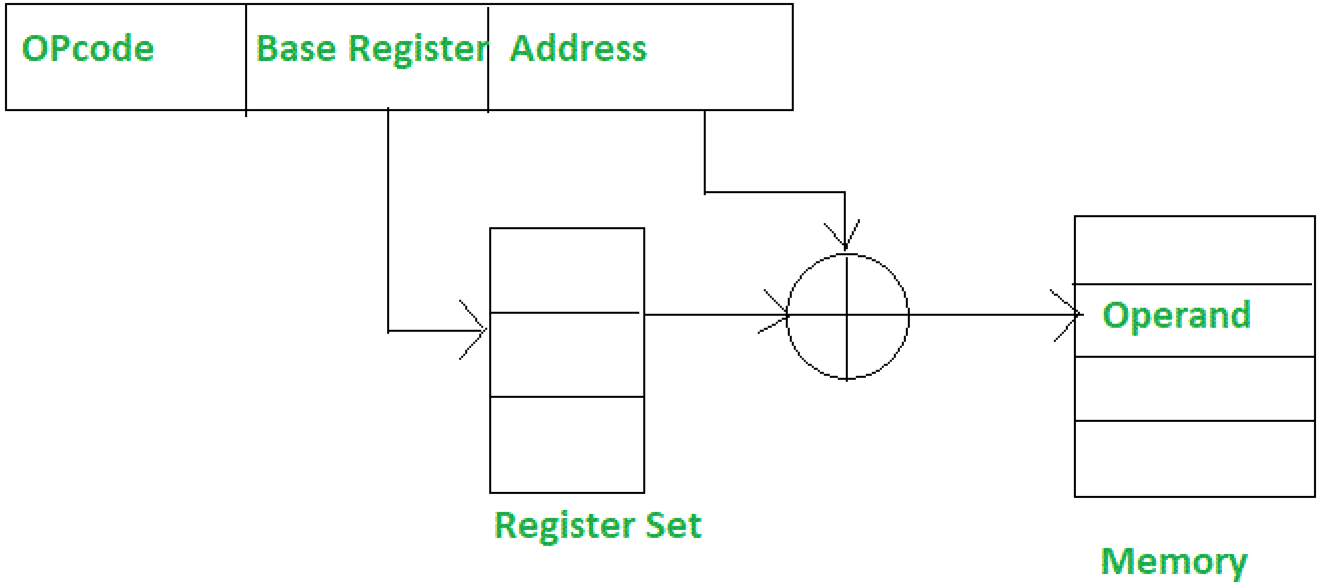




RELATIVE ADDRESSING MODE

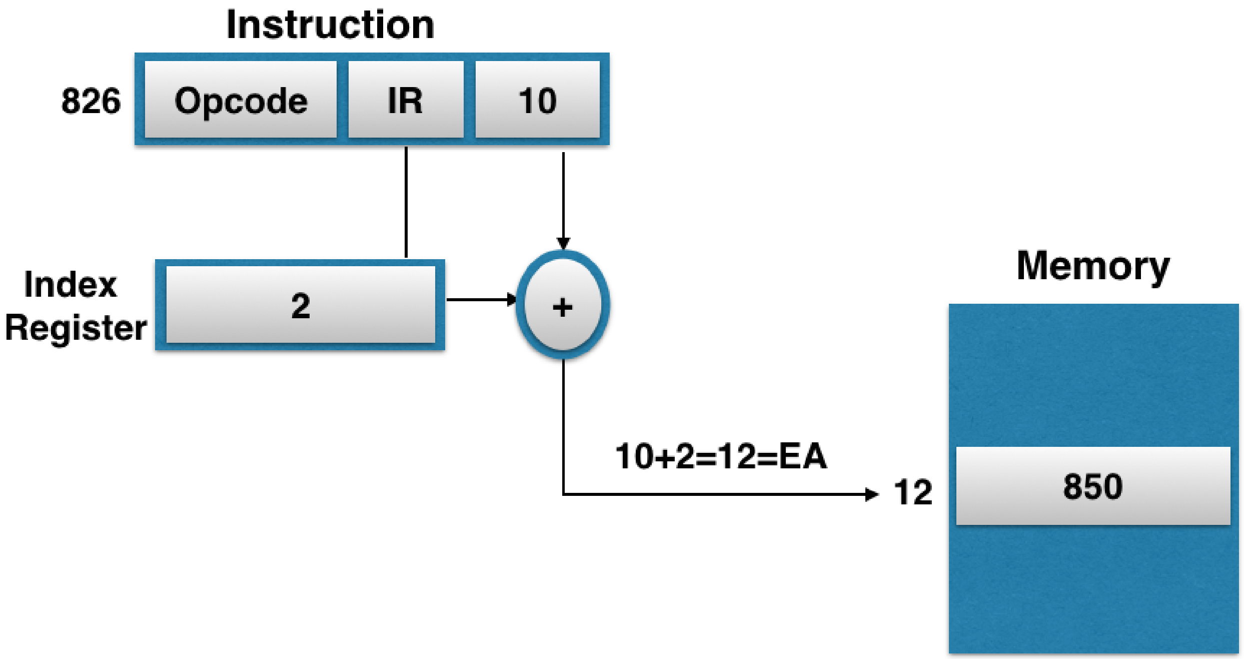
Base-Register Addressing

- A holds displacement
- R holds pointer to base address
- R may be explicit or implicit
- e.g. segment registers in 80x86



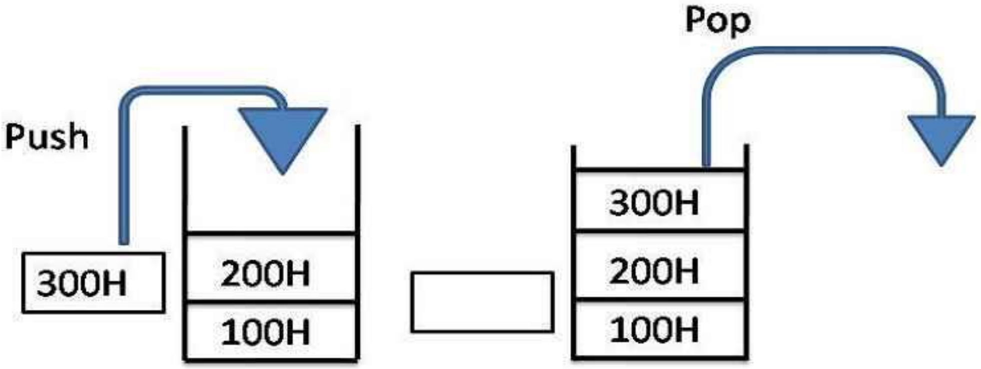
Indexed Addressing

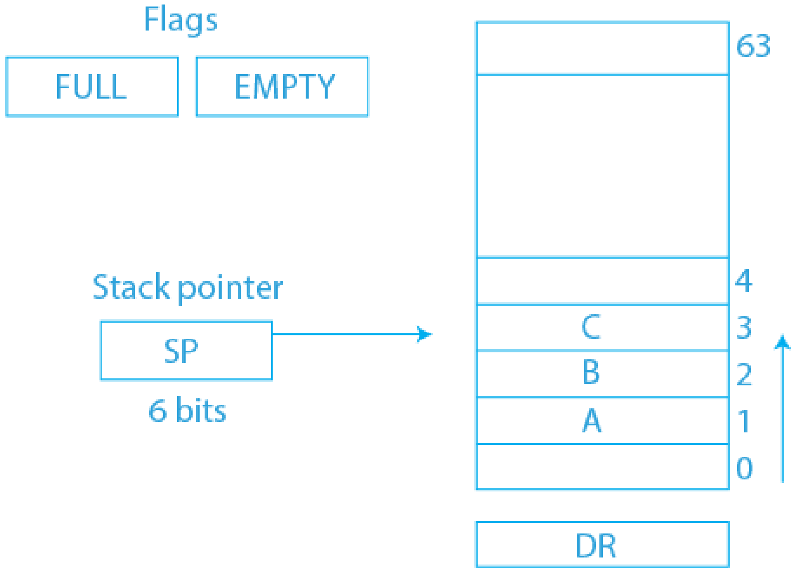
- A = base
- R = displacement
- EA = A + R
- Good for accessing arrays
  - EA = A + R
  - R++



Stack Addressing

- Operand is (implicitly) on top of stack
- Use Special Register => Stack Pointer
- e.g.
  - ADD (SP) Pop top two items from stack and add



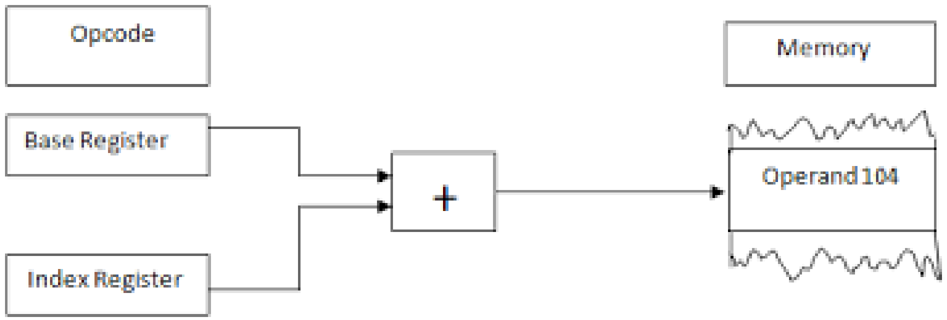


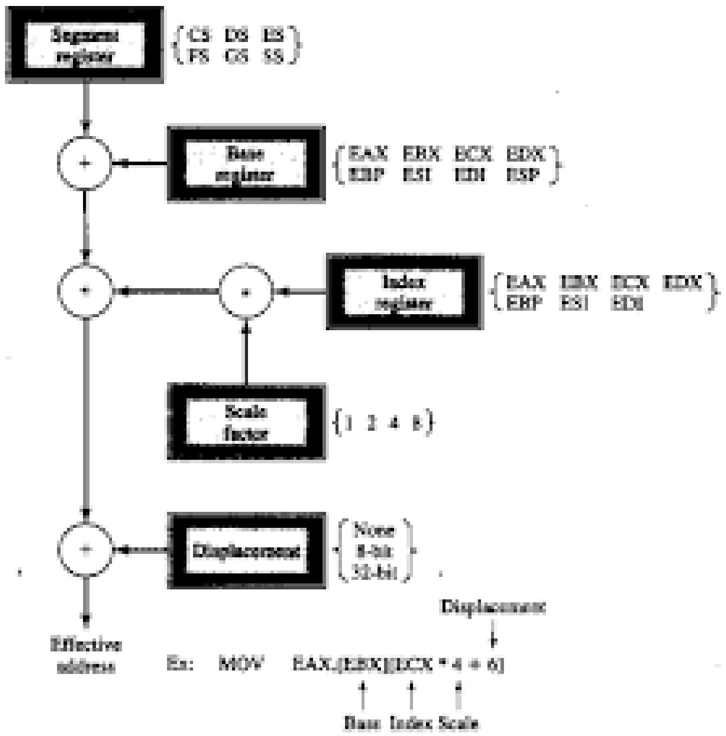
Combinations

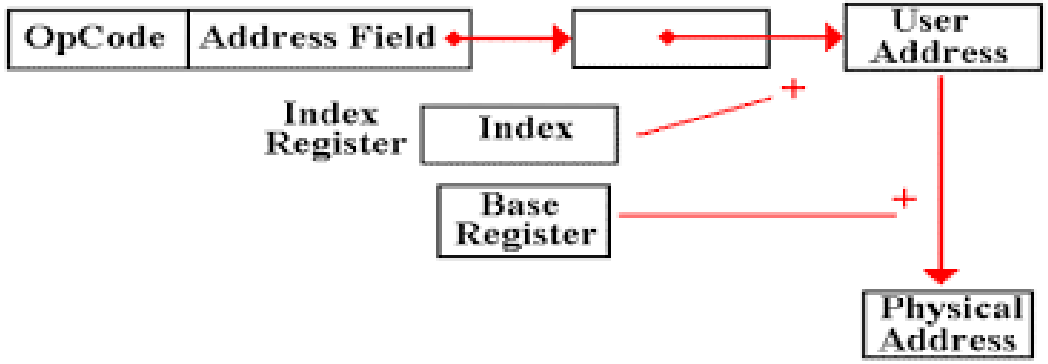
- Post-index
- EA = (A) + (R)

- Pre-index
- EA = (A+(R))

- (Draw the diagrams)

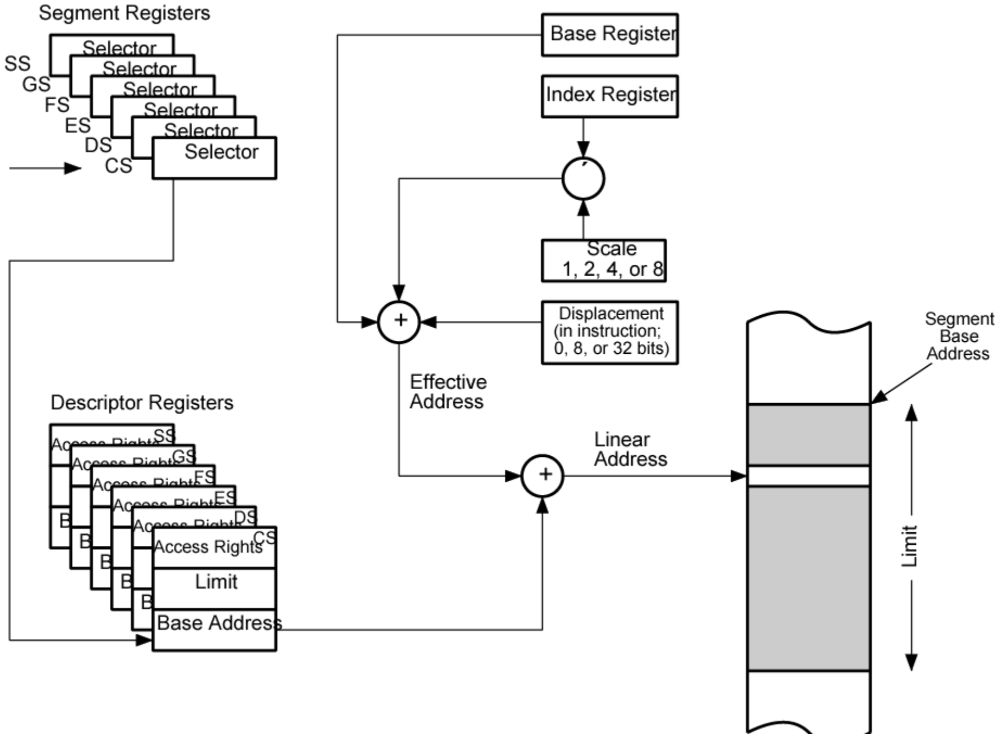






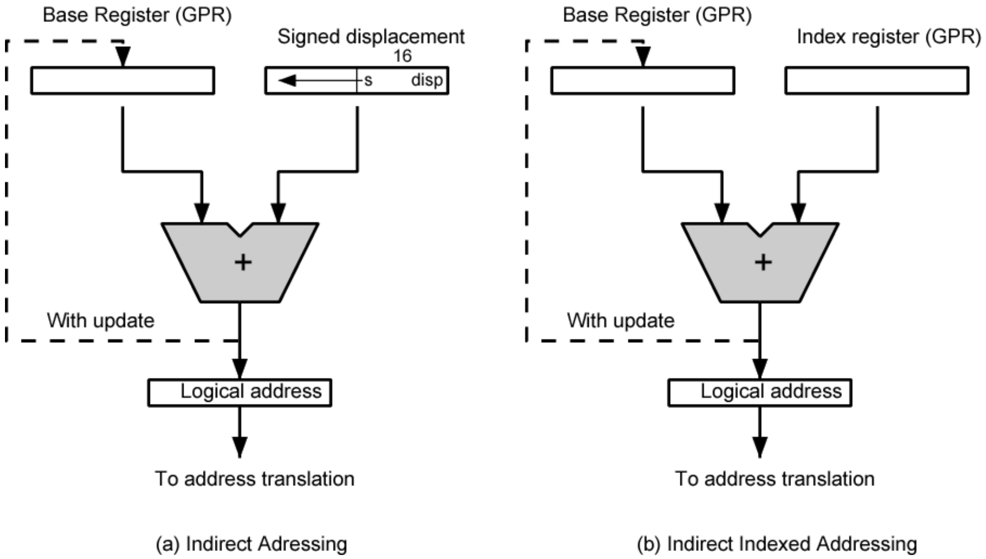
Example: Pentium Addressing Modes

- Virtual or effective address is offset into segment
  - Starting address plus offset gives linear address
  - This goes through page translation if paging enabled
- 12 addressing modes available
  - Immediate
  - Register operand
  - Displacement
  - Base
  - Base with displacement
  - Scaled index with displacement
  - Base with index and displacement
  - Base scaled index with displacement
  - Relative



Example: PowerPC Addressing Modes

- Load/store architecture
  - Indirect
    - Instruction includes 16 bit displacement to be added to base register (may be GP register)
    - Can replace base register content with new address
  - Indirect indexed
    - Instruction references base register and index register (both may be GP)
    - EA is sum of contents
- Branch address
  - Absolute
  - Relative
  - Indirect
- Arithmetic
  - Operands in registers or part of instruction
  - Floating point is register only

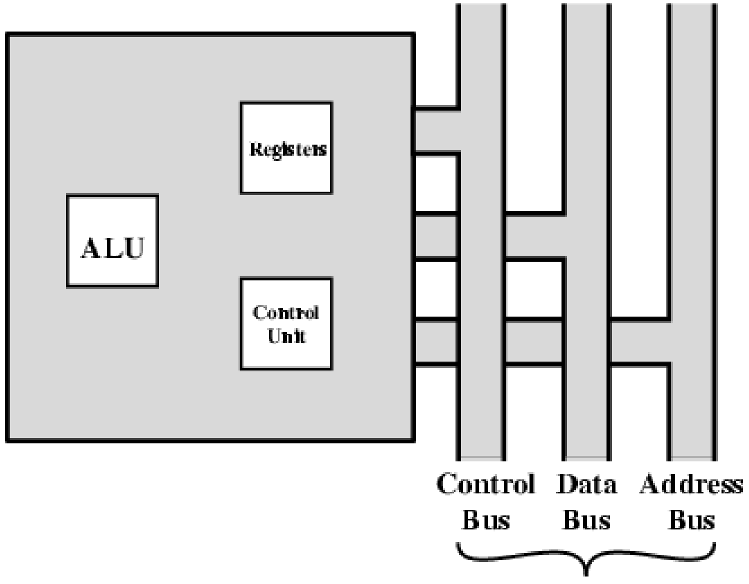


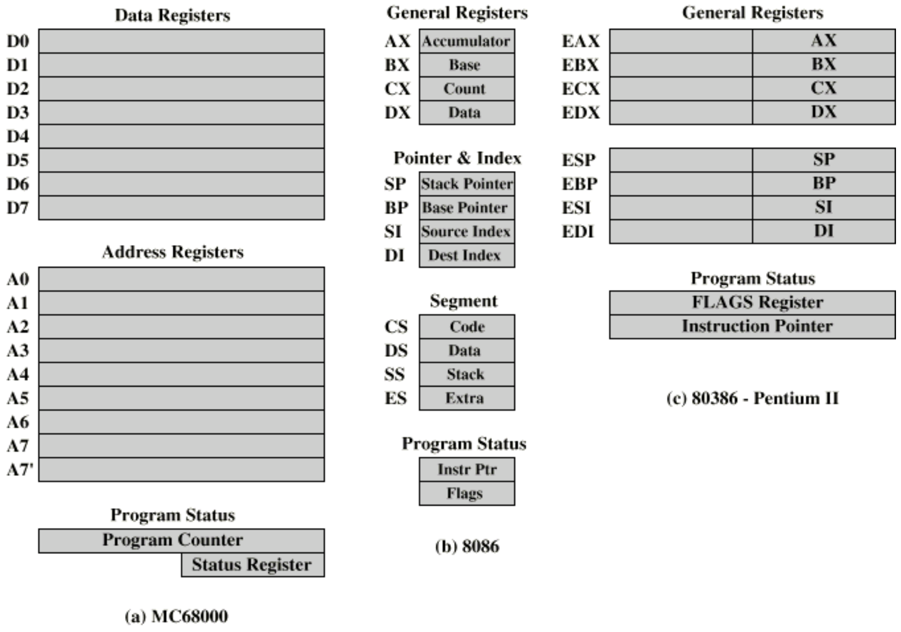
(a) Indirect Adressing

(b) Indirect Indexed Addressing

Register Model

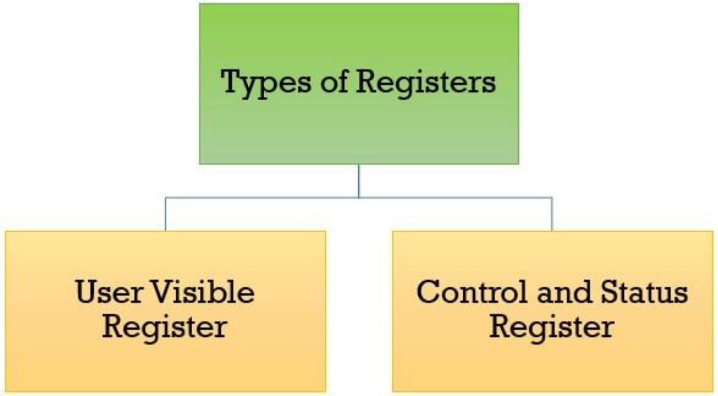
- CPU must have some working space (temporary storage)
- Called registers
- Number and function vary between processor designs
- One of the major design decisions
- Top level of memory hierarchy
- Registers must be named
  - e.g. A, B, R00, R01, PC, etc.

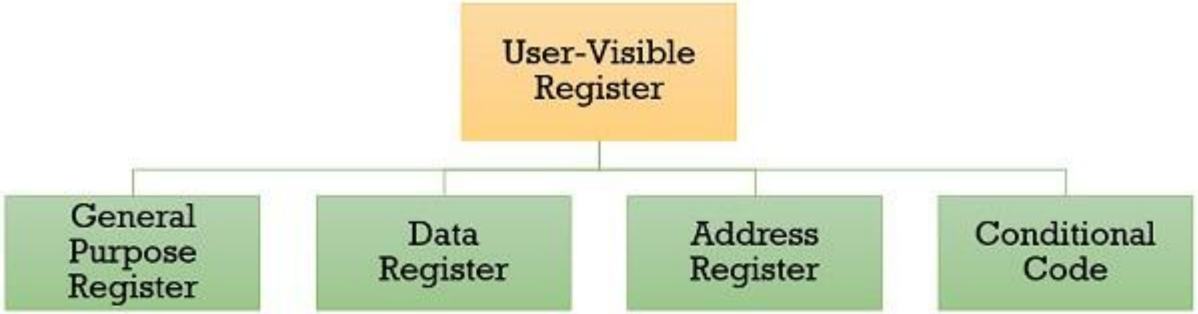




Register Model

- User Visible Registers
  - General Purpose
  - Data
  - Address
  - Condition Codes
  - May be true general purpose
  - May be restricted
  - May be used for data or addressing
  - Data
    - Accumulator
  - Addressing
    - Segment
  - Make them general purpose
    - Increase flexibility and programmer options
    - Increase instruction size & complexity
  - Make them specialized
    - Smaller (faster) instructions
    - Less flexibility

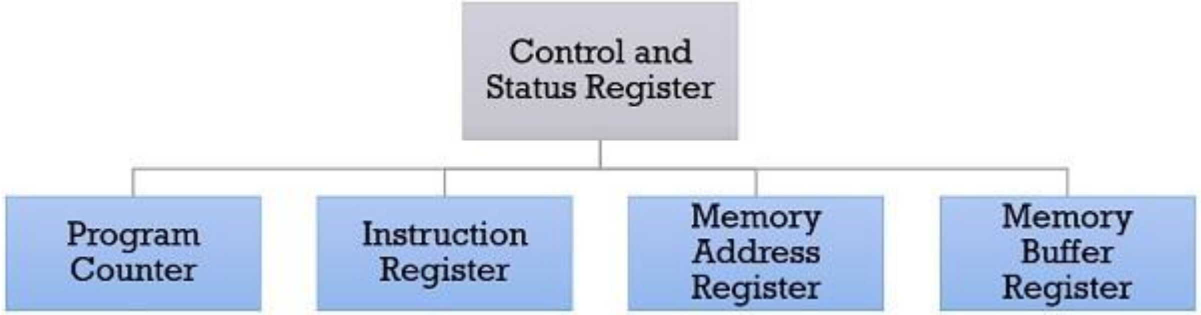




- Condition Code Registers
  - Sets of individual bits
    - e.g. result of last operation was zero
  - Can be read (implicitly) by programs
    - e.g. Jump if zero
  - Can not (usually) be set by programs

Register Model

- Control & Status Registers
  - Program Counter
  - Instruction Decoding Register
  - Memory Address Register
  - Memory Buffer Register
- Program Status Word
  - A set of bits
  - Includes Condition Codes
  - Sign of last result
  - Zero
  - Carry
  - Equal
  - Overflow
  - Interrupt enable/disable
  - Supervisor

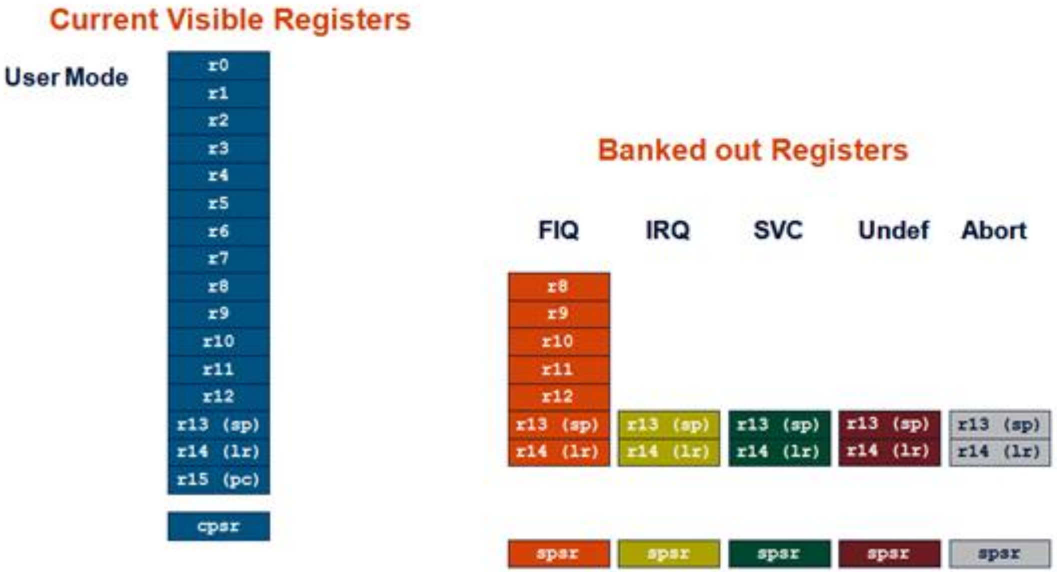


Supervisor Mode or Kernel mode

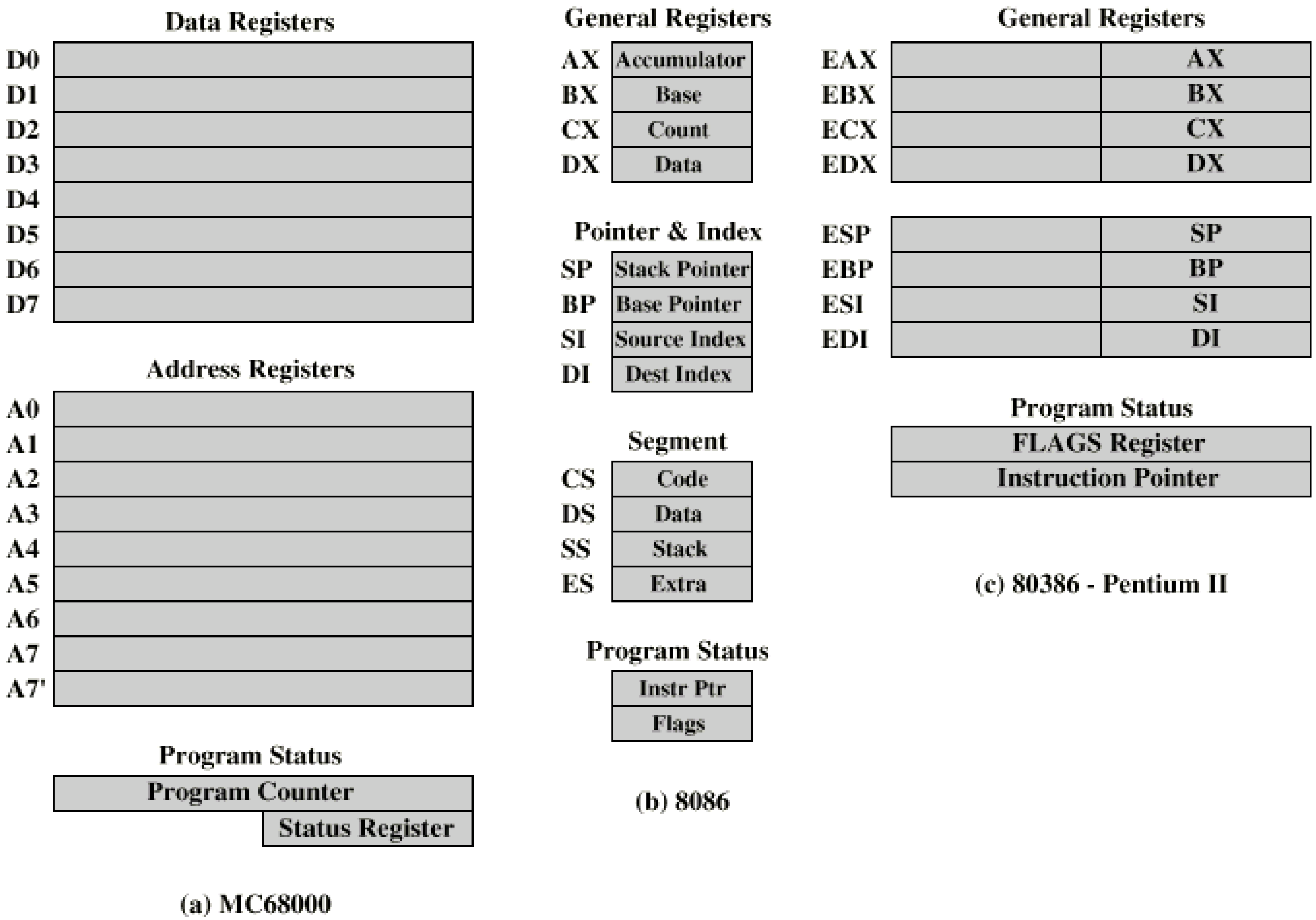
- CPU Mode
- Allows privileged instructions to execute
- Can access all resources
- Used by operating system
- Not available to user programs

Register Model

- Other Registers
  - May have registers pointing to:
    - Process control blocks (see O/S)
    - Interrupt Vectors (see O/S)
  - Special Registers
- Depend on CPU Design
- Must be identified



Example Register Organizations



Data Registers

D0
D1
D2
D3
D4
D5
D6
D7

Address Registers

A0
A1
A2
A3
A4
A5
A6
A7
A7'

Program Status

Program Counter
Status Register

(a) MC68000

General Registers

AX	Accumulator
BX	Base
CX	Count
DX	Data

Pointer & Index

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Dest Index

Segment

CS	Code
DS	Data
SS	Stack
ES	Extra

Program Status

Instr Ptr
Flags

(b) 8086

General Registers

EAX		AX
EBX		BX
ECX		CX
EDX		DX

ESP		SP
EBP		BP
ESI		SI
EDI		DI

Program Status

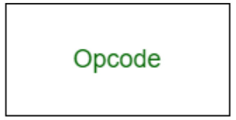
FLAGS Register
Instruction Pointer

(c) 80386 - Pentium II

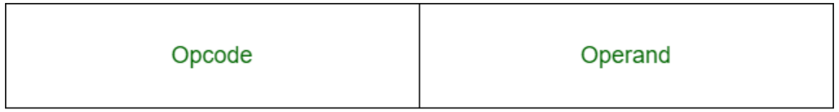
Instruction Format

Elements of an Instruction

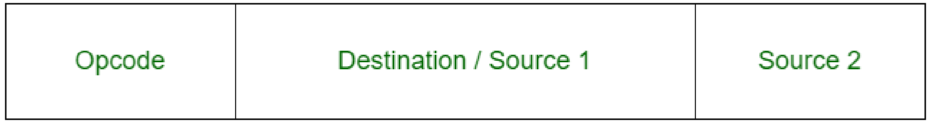
- Operation code (Op-code) – Must have
    - Do this
    - Command / What to do
- Associated Data
    - Called “Operand”
    - May be Zero (NO) or One or Two or Three or More
        - Depend on Instruction
    - Normally specify
        - Source of Data, One or Two
        - Destination or Result of Data
        - Next Instruction reference
    - Source Operand reference
        - To this
    - Result Operand reference
        - Put the answer here
    - Next Instruction Reference
        - When you have done that, do this…



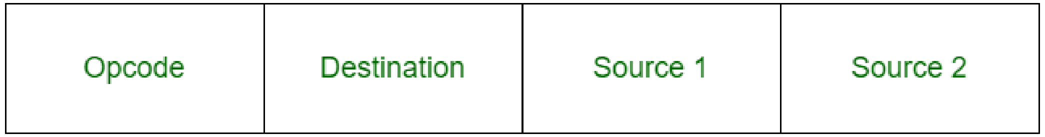
0-address Instruction Format



1-address Instruction Format



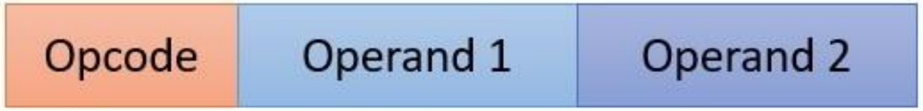
2-address Instruction Format



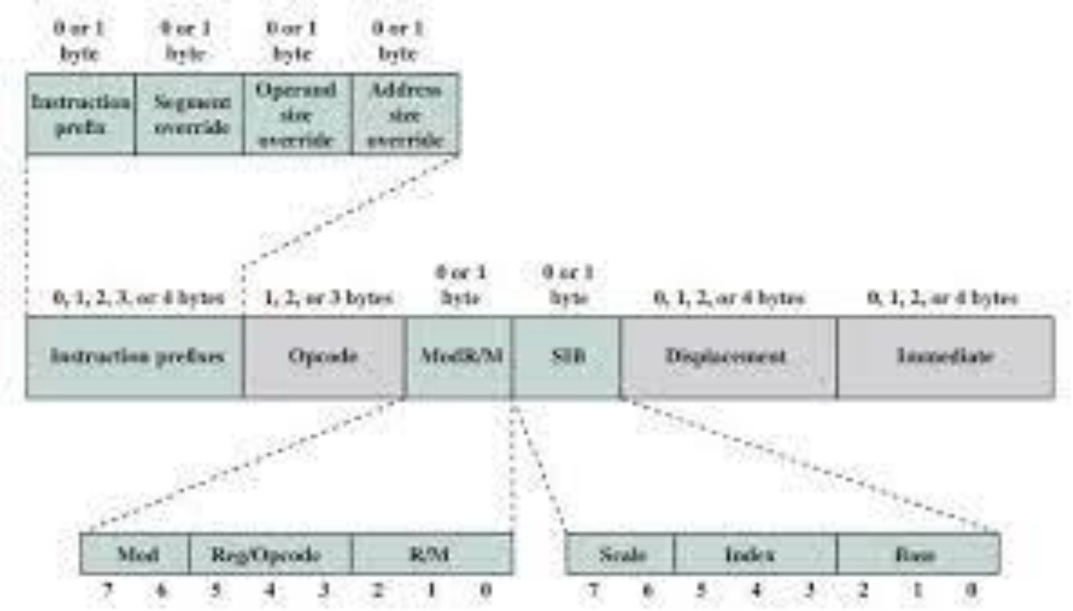
3-address Instruction Format

Operands

- Specify data for instruction
- Tell where is data
  - => Addressing Mode
- Data Location
  - CPU Register
  - Main memory (or virtual memory or cache)
  - I/O Device
  - Coded in the instruction



Instruction Format



Instruction Representation

- In machine code each instruction has a unique bit pattern

  => Binary Number

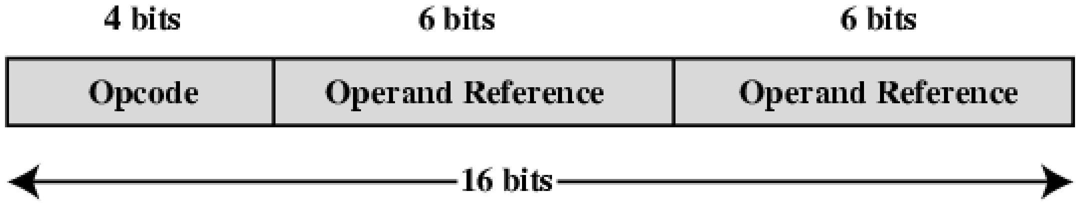
- For human consumption (well, programmers anyway) a symbolic representation is used

  => Assembly Language / Code

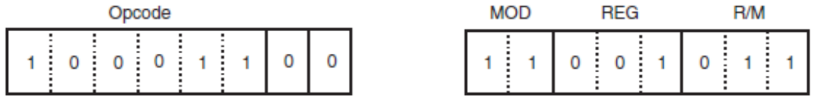
  e.g. ADD, SUB, LOAD

- Operands can also be represented in this way
  - ADD A,B

- Easily Mapping between Machine Code and Assembly Code
  - Opcode => Binary ⇔ Operation Symbolic
  - Operand => Binary ⇔ Operand Symbolic

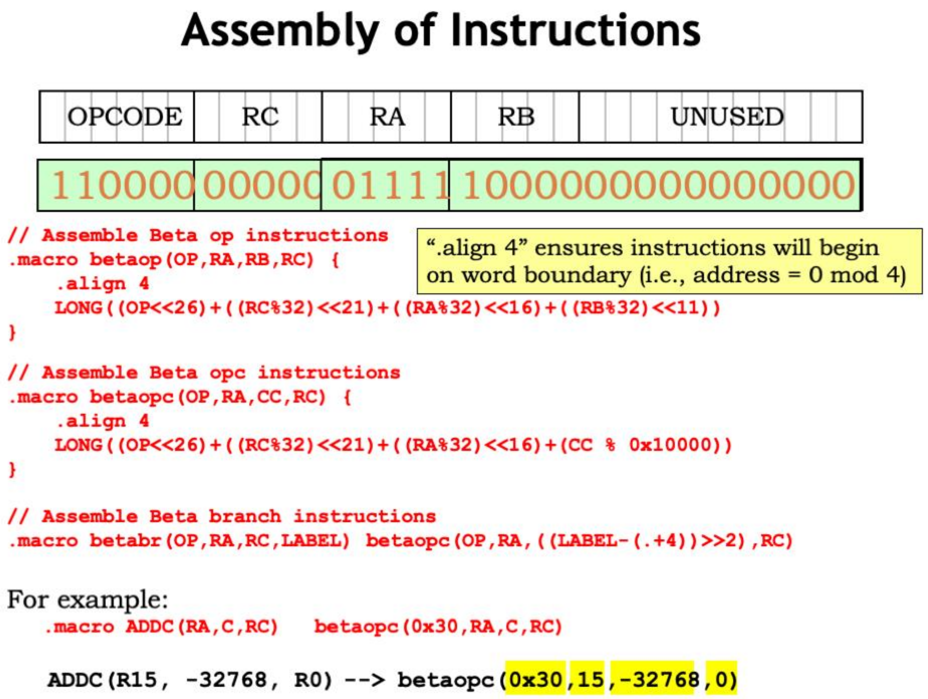


MOV BX , CS



Opcode = MOV
MOD = R/M is a register
REG = CS
R/M = BX

Instruction Representation



Assembly of Instructions

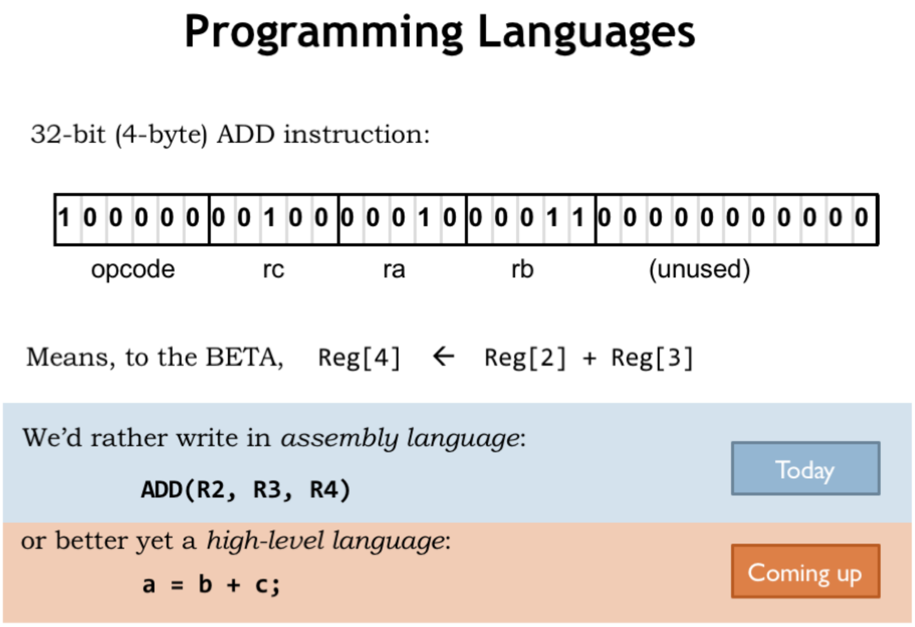
OPCODE	RC	RA	RB	UNUSED
110000	00000	01111	1000000000000000	

```
// Assemble Beta op instructions
.macro betaop(OP,RA,RB,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+((RB%32)<<11))
}

// Assemble Beta opc instructions
.macro betaopc(OP,RA,CC,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+(CC % 0x10000))
}

// Assemble Beta branch instructions
.macro betabr(OP,RA,RC,LABEL) betaopc(OP,RA,((LABEL-(.+4))>>2),RC)
```

".align 4" ensures instructions will begin on word boundary (i.e., address = 0 mod 4)

For example:

```
   .macro ADDC(RA,C,RC)   betaopc(0x30,RA,C,RC)

   ADDC(R15, -32768, R0) --> betaopc(0x30,15,-32768,0)
```



Programming Languages

32-bit (4-byte) ADD instruction:

1 0 0 0 0 0	0 0 1 0 0	0 0 0 1 0	0 0 0 1 1	0 0 0 0 0 0 0 0 0 0 0
opcode	rc	ra	rb	(unused)

Means, to the BETA, `Reg[4] ← Reg[2] + Reg[3]`

We'd rather write in *assembly language*:

```
ADD(R2, R3, R4)
```

Today

or better yet a *high-level language*:

```
a = b + c;
```

Coming up

Instruction Codes

- Low Level
  - Machine Code

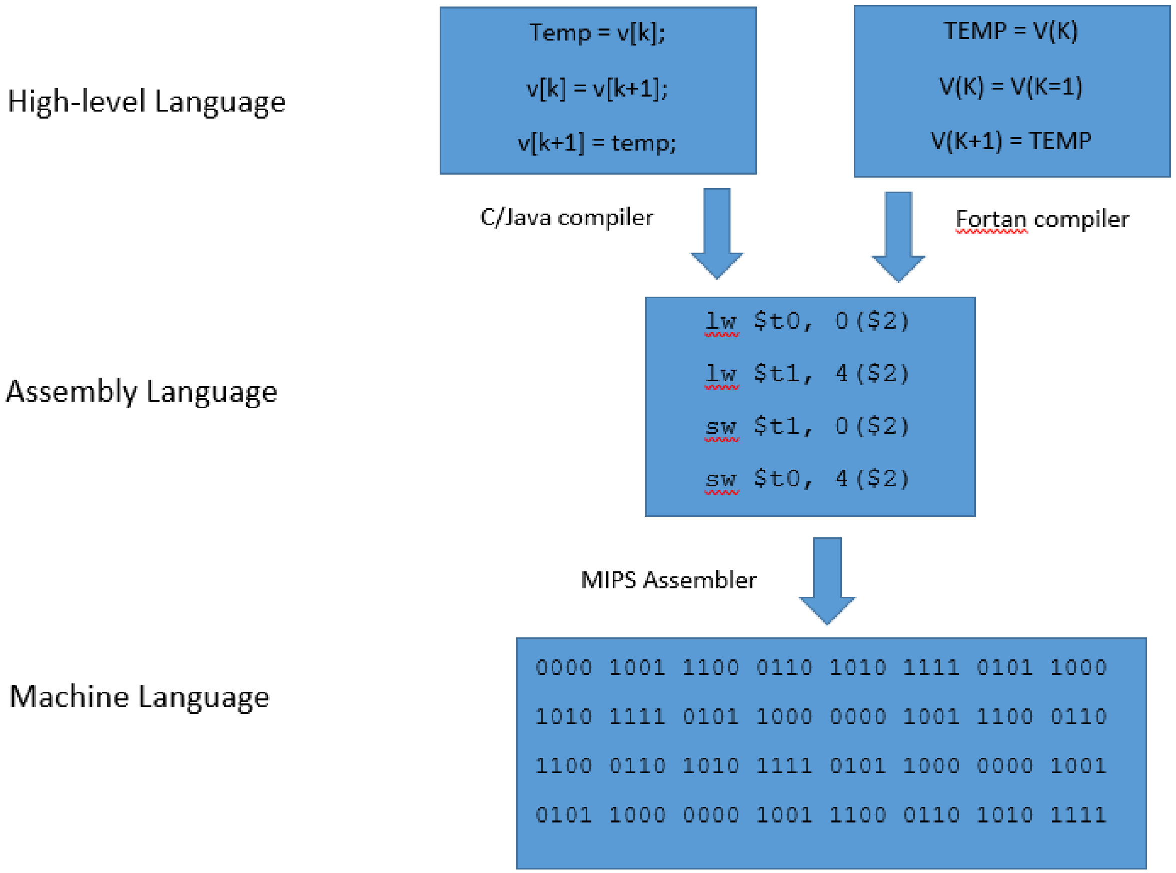
    => Binary

    => Computer uses
  - Assembly Code

    => Symbolic Represent for Machine Code

    => Human uses
- Intermediate / High Level

  => Human uses
  - C/C++ Language
  - Other Language
  - Script-Based Language
    - Collections of Programs



High-level Language

```
Temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
TEMP = V(K)
V(K) = V(K=1)
V(K+1) = TEMP
```

C/Java compiler

Fortan compiler

Assembly Language

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

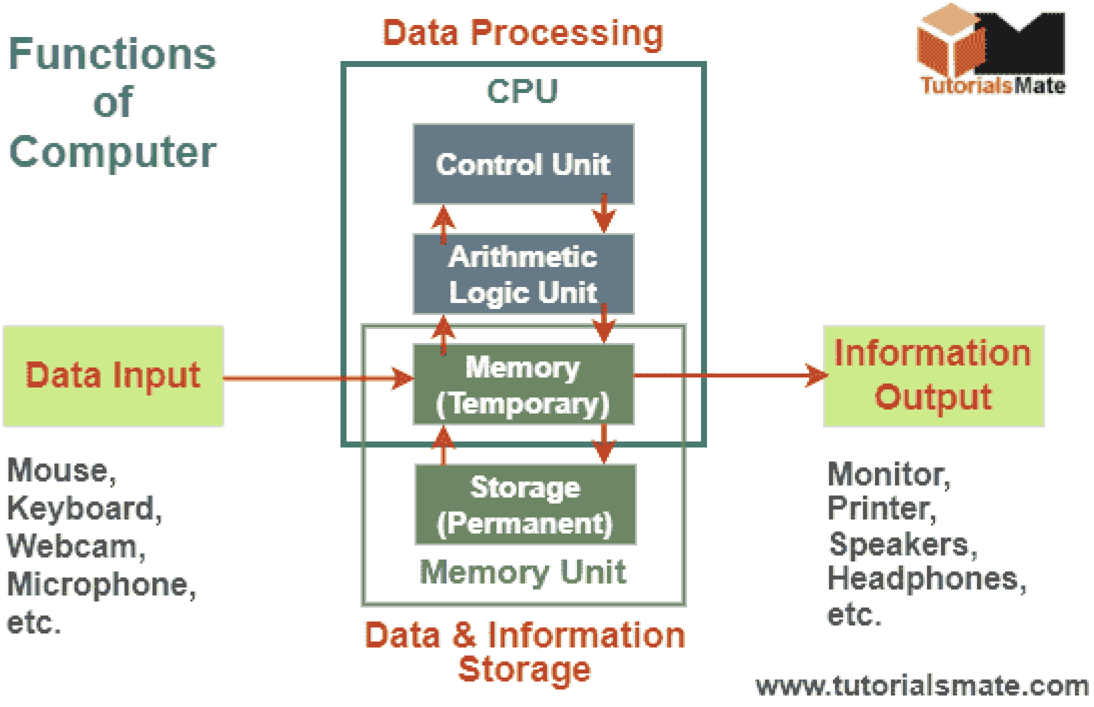
MIPS Assembler

Machine Language

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

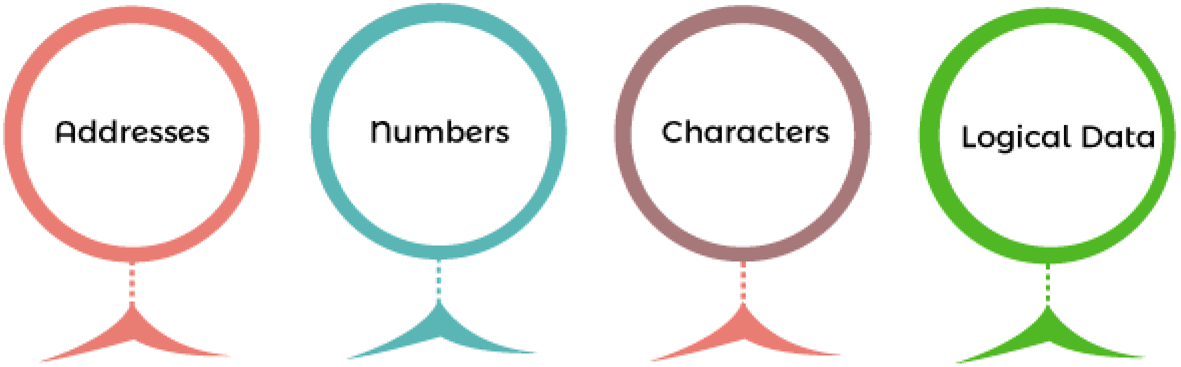
Instruction Types

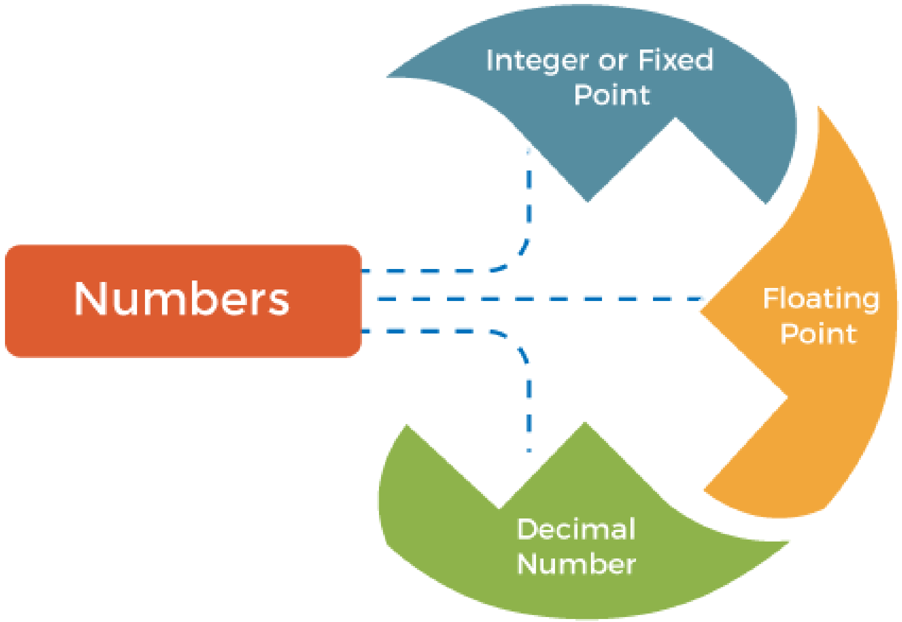
- Depend on “Opcode”
- Working with CPU Registers
- Data processing
  - Process numbers => Change Values
  - Using ALU / Operating Unit
- Data storage (main memory)
  - Store Numbers
  - Using Main Memory
- Data movement (I/O)
  - Move/Transfer Data
  - Using Input / Output
- Program flow control
  - Change Sequence of Program
  - Location of Next Instruction to be executed



Types of Operand

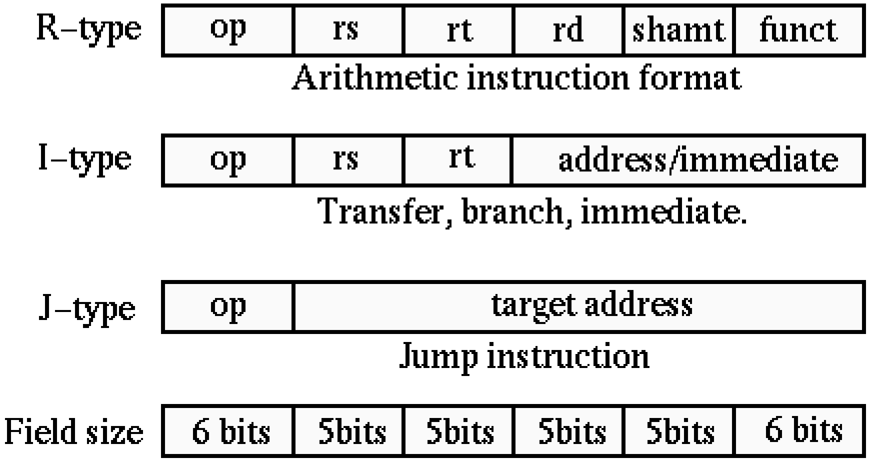
- Addresses
- Numbers
  - Integer
  - Floating Point
- Characters
  - ASCII etc.
- Logical Data
  - Bits or flags



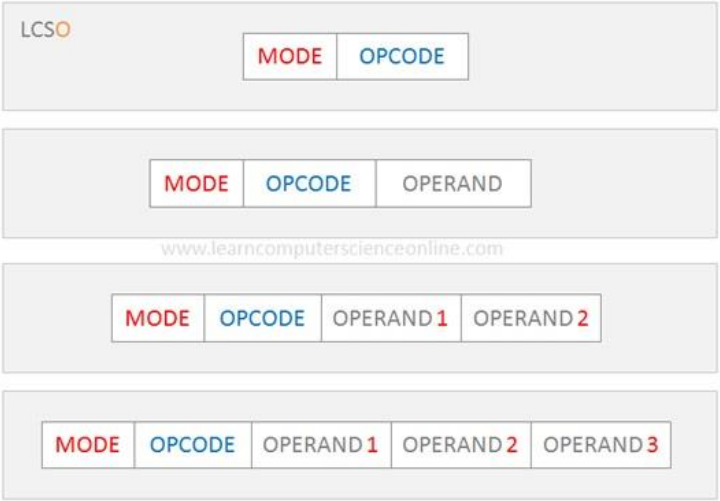


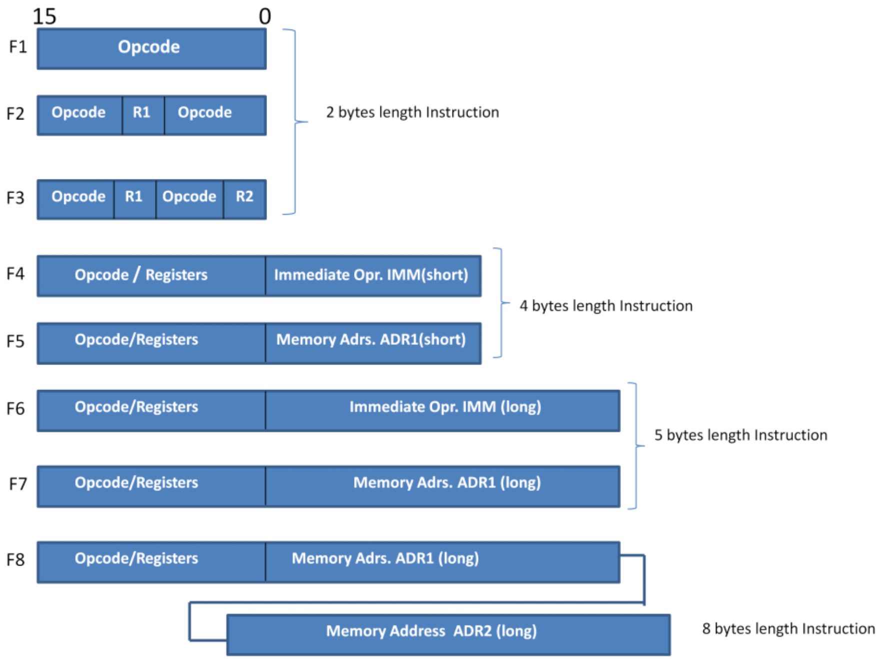
Instruction Formats

- Layout of bits in an instruction
- Includes opcode
- Includes (implicit or explicit) operand(s)
- Usually more than one instruction format in an instruction set



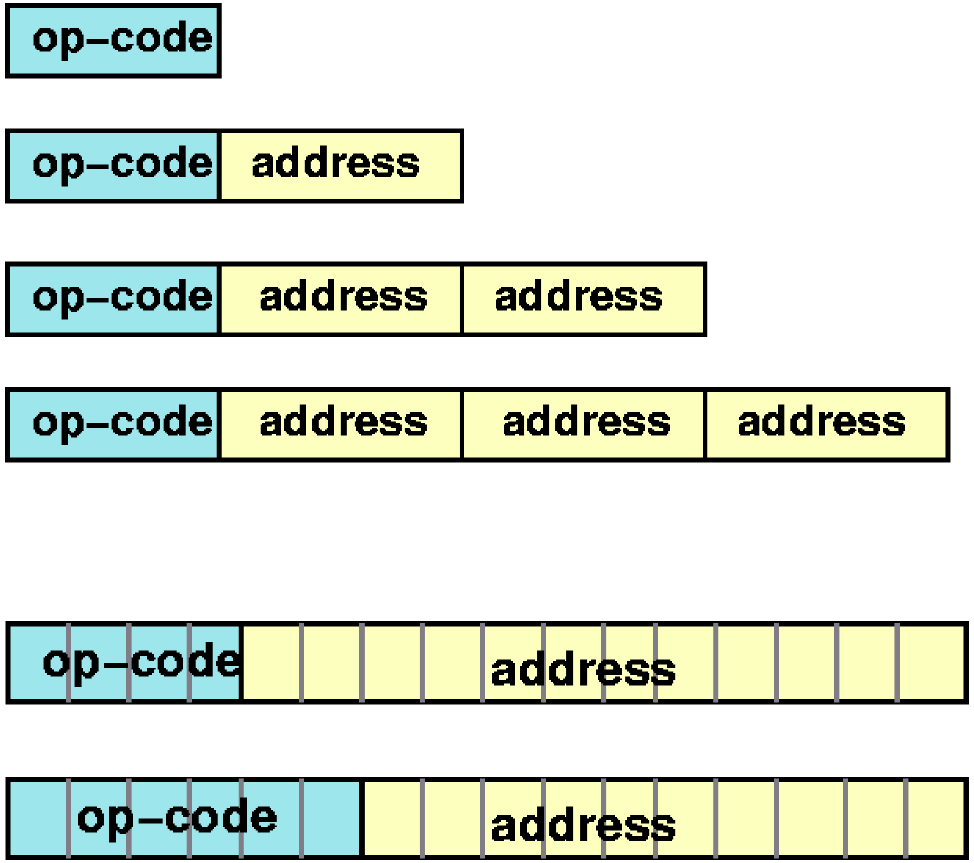
R–type	op	rs	rt	rd	shamt	funct
	Arithmetic instruction format					
I–type	op	rs	rt	address/immediate		
	Transfer, branch, immediate.					
J–type	op	target address				
	Jump instruction					
Field size	6 bits	5bits	5bits	5bits	5bits	6 bits


LCSO
MODE | OPCODE
MODE | OPCODE | OPERAND
www.learncomputerscienceonline.com
MODE | OPCODE | OPERAND 1 | OPERAND 2
MODE | OPCODE | OPERAND 1 | OPERAND 2 | OPERAND 3

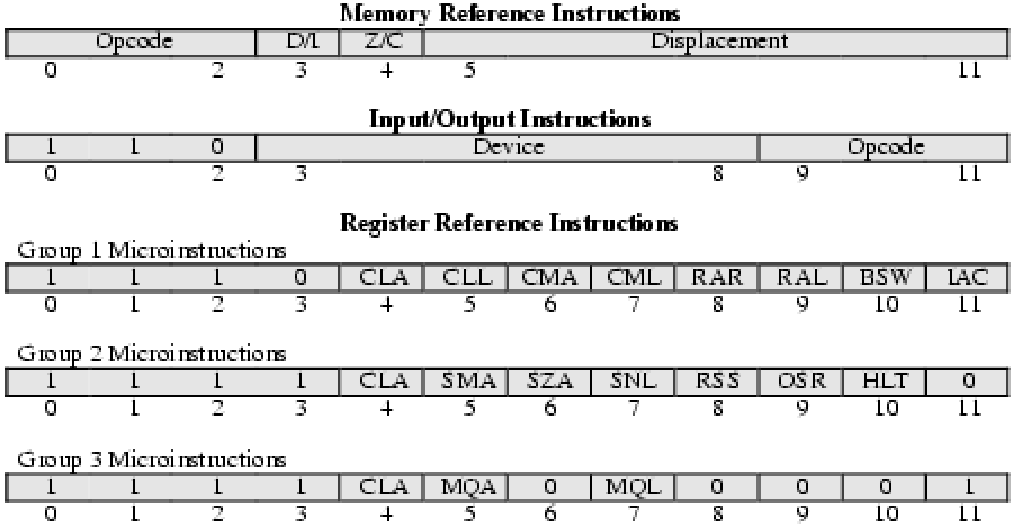


Instruction Length

- Affected by and affects:
  - Memory size
  - Memory organization
  - Bus structure
  - CPU complexity
  - CPU speed
- Trade off between powerful instruction repertoire and saving space
- Allocation of Bits
  - Number of addressing modes
  - Number of operands
  - Register versus memory
  - Number of register sets
  - Address range
  - Address granularity



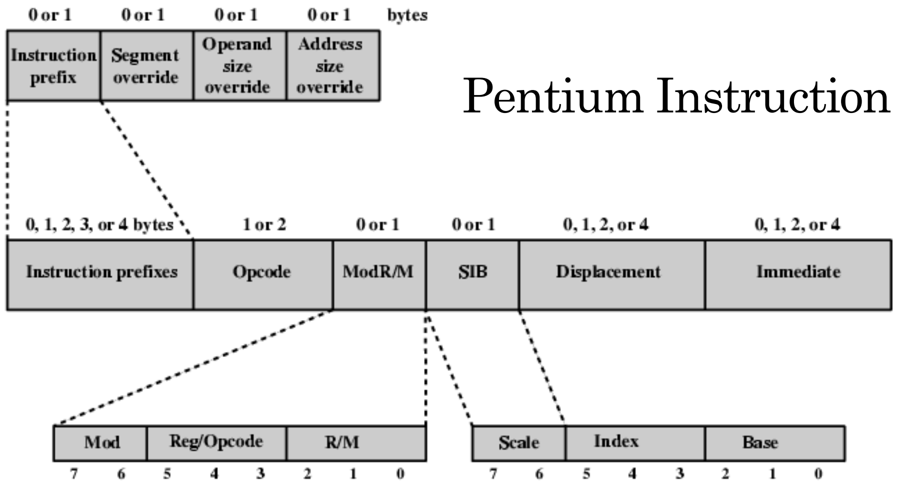
Instruction Format Examples


Memory Reference Instructions
Opcode
D/I
Z/C
Displacement
0 2 3 4 5 11
Input/Output Instructions
1 1 0
Device
Opcode
0 2 3 8 9 11
Register Reference Instructions
Group 1 Microinstructions
1 1 1 0 CLA CLL CMA CML RAR RAL BSW IAC
0 1 2 3 4 5 6 7 8 9 10 11
Group 2 Microinstructions
1 1 1 1 CLA SMA SZA SNL RSS OSR HLT 0
0 1 2 3 4 5 6 7 8 9 10 11
Group 3 Microinstructions
1 1 1 1 CLA MQA 0 MQL 0 0 0 1
0 1 2 3 4 5 6 7 8 9 10 11

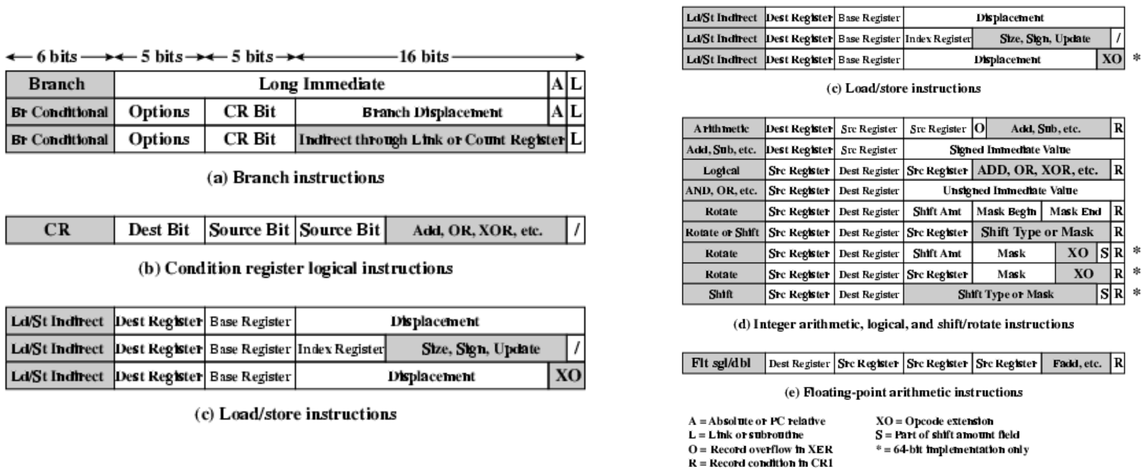
D/I = Direct/Indirect address
Z/C = Page 0 or Current page
CLA = Clear Accumulator
CLL = Clear Link
CMA = CoMplement Accumulator
CML = CoMplement Link
RAR = Rotate Accumulator Right
RAL = Rotate Accumulator Left
BSW = Byte SWap

IAC = Increment ACcumulator
SMA = Skip on Minus Accumulator
SZA = Skip on Zero Accumulator
SNL = Skip on Nonzero Link
RSS = Reverse Skip Sense
OSR = Or with Switch Register
HLT = HaLT
MQA = Multiplier Quotient into Accumulator
MQL = Multiplier Quotient Load

PDP-8 Instruction Format



Pentium Instruction Format



(a) Branch instructions

(b) Condition register logical instructions

(c) Load/store instructions

(c) Load/store instructions

(d) Integer arithmetic, logical, and shift/rotate instructions

(e) Floating-point arithmetic instructions

A = Absolute or PC relative
L = Link or subroutine
O = Record overflow in XER
R = Record condition in CR1

XO = Opcode extension
S = Part of shift amount field
* = 64-bit implementation only

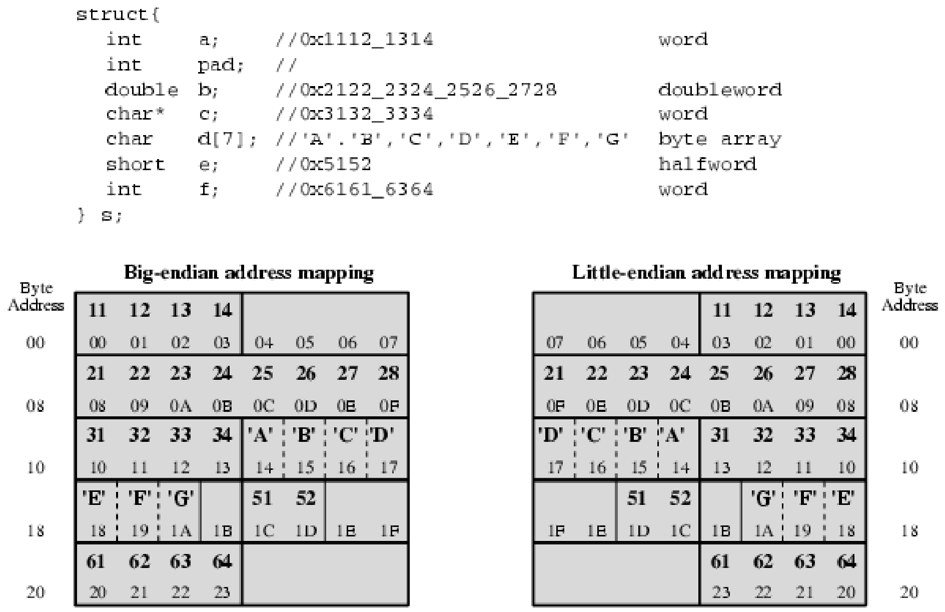
PowerPC Instruction Formats

Byte Order

- Binary Arrangement
- What order do we read numbers that occupy more than one byte
- e.g. (numbers in hex to make it easy to read)
- 12345678 can be stored in 4x8bit locations as follows

Address	Value (1)	Value(2)
184	12	78
185	34	56
186	56	34
186	78	12

- i.e. read top down or bottom up?



```
struct{
   int     a;     //0x1112_1314                      word
   int     pad;   //
   double  b;     //0x2122_2324_2526_2728            doubleword
   char*   c;     //0x3132_3334                      word
   char    d[7];  //'A'.'B','C','D','E','F','G'      byte array
   short   e;     //0x5152                           halfword
   int     f;     //0x6161_6364                      word
} s;
```

Big-endian address mapping

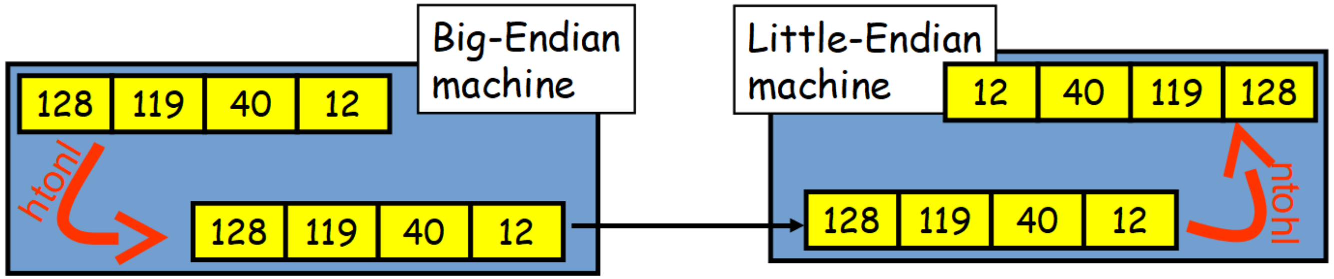
Byte Address								
	11	12	13	14				
00	00	01	02	03	04	05	06	07
	21	22	23	24	25	26	27	28
08	08	09	0A	0B	0C	0D	0E	0F
	31	32	33	34	'A'	'B'	'C'	'D'
10	10	11	12	13	14	15	16	17
	'E'	'F'	'G'		51	52		
18	18	19	1A	1B	1C	1D	1E	1F
	61	62	63	64				
20	20	21	22	23				

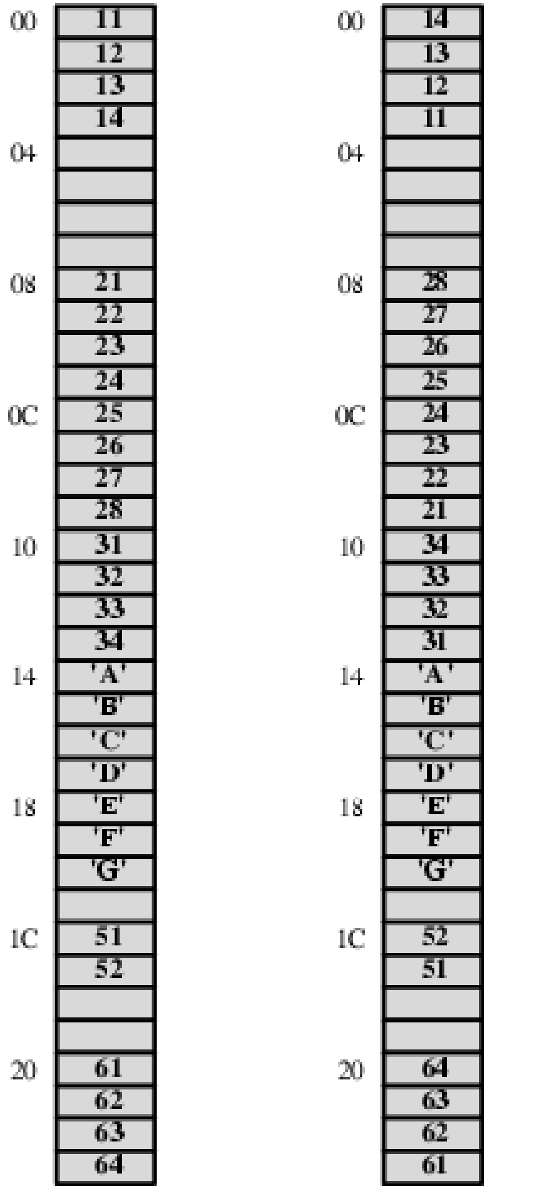
Little-endian address mapping

								Byte Address
				11	12	13	14	
07	06	05	04	03	02	01	00	00
21	22	23	24	25	26	27	28	
0F	0E	0D	0C	0B	0A	09	08	08
'D'	'C'	'B'	'A'	31	32	33	34	
17	16	15	14	13	12	11	10	10
		51	52		'G'	'F'	'E'	
1F	1E	1D	1C	1B	1A	19	18	18
				61	62	63	64	
				23	22	21	20	20

Byte Order Names

- The problem is called Endian
- The system on the left has the least significant byte in the lowest address
- This is called “Big-Endian”
- The system on the right has the least significant byte in the highest address
- This is called “Little-Endian”





00	11	00	14
	12		13
	13		12
	14		11
04		04	
08	21	08	28
	22		27
	23		26
	24		25
0C	25	0C	24
	26		23
	27		22
	28		21
10	31	10	34
	32		33
	33		32
	34		31
14	'A'	14	'A'
	'B'		'B'
	'C'		'C'
	'D'		'D'
18	'E'	18	'E'
	'F'		'F'
	'G'		'G'
1C	51	1C	52
	52		51
20	61	20	64
	62		63
	63		62
	64		61

(a) Big-endian

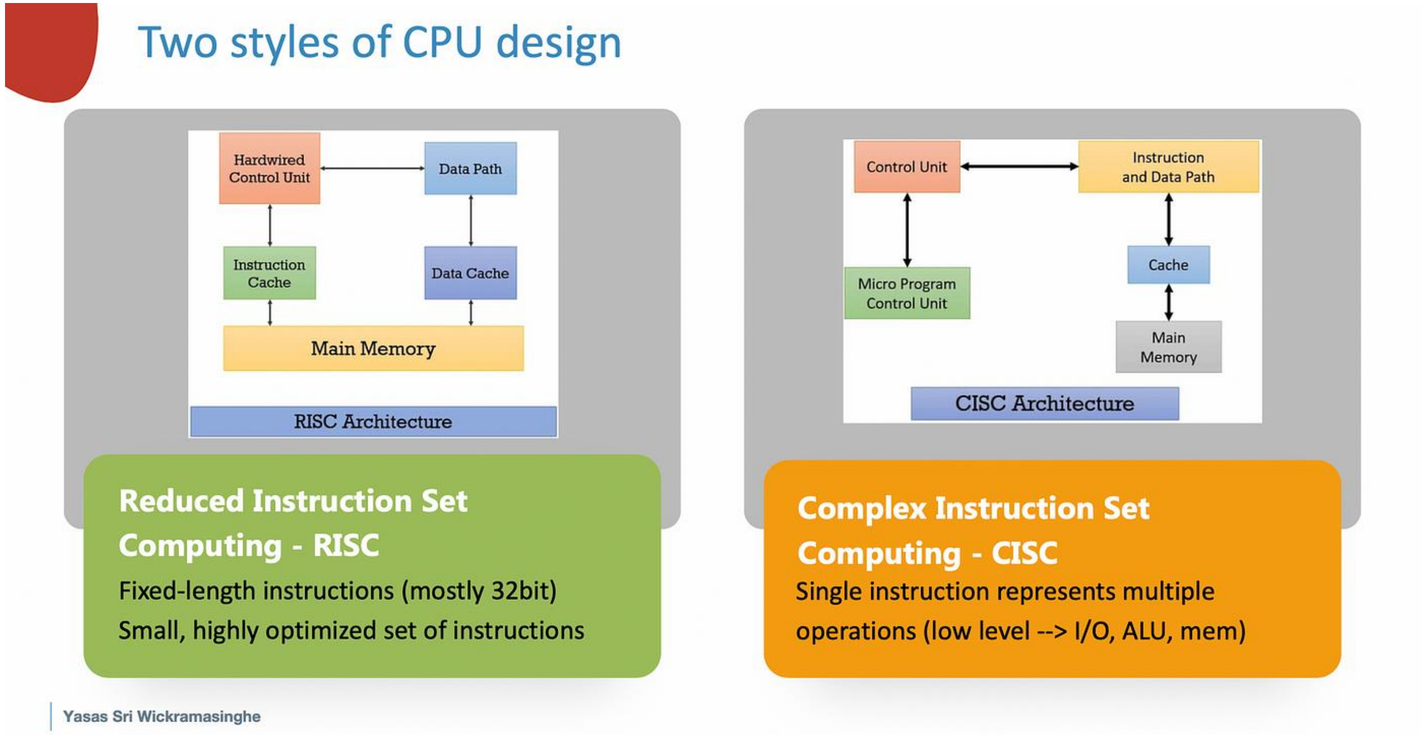
(b) Little-endian

Standard…What Standard?

- Pentium (80x86), VAX are little-endian
- IBM 370, Motorola 680x0 (Mac), and most RISC are big-endian
- Internet is big-endian
  - Makes writing Internet programs on PC more awkward!
  - WinSock provides htoi and itoh (Host to Internet & Internet to Host) functions to convert

Types of Computer Architecture

- Different Concepts
- RISC – Reduced Instruction Set Computer
- CISC – Complex Instruction Set Computer



Two styles of CPU design

Reduced Instruction Set Computing - RISC

Fixed-length instructions (mostly 32bit)

Small, highly optimized set of instructions

Complex Instruction Set Computing - CISC

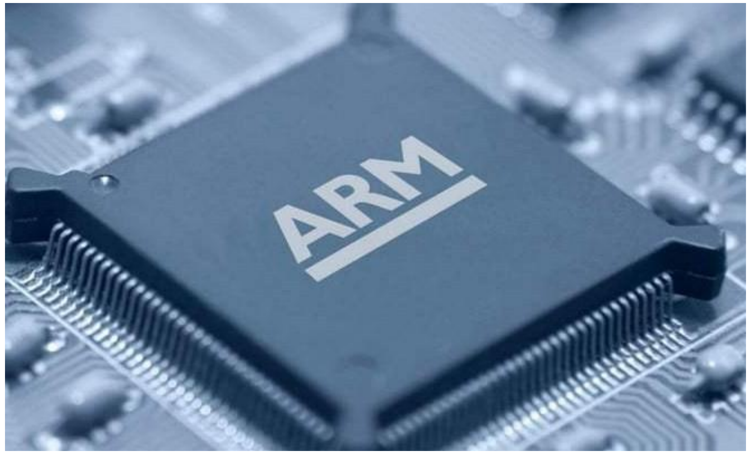
Single instruction represents multiple operations (low level --> I/O, ALU, mem)

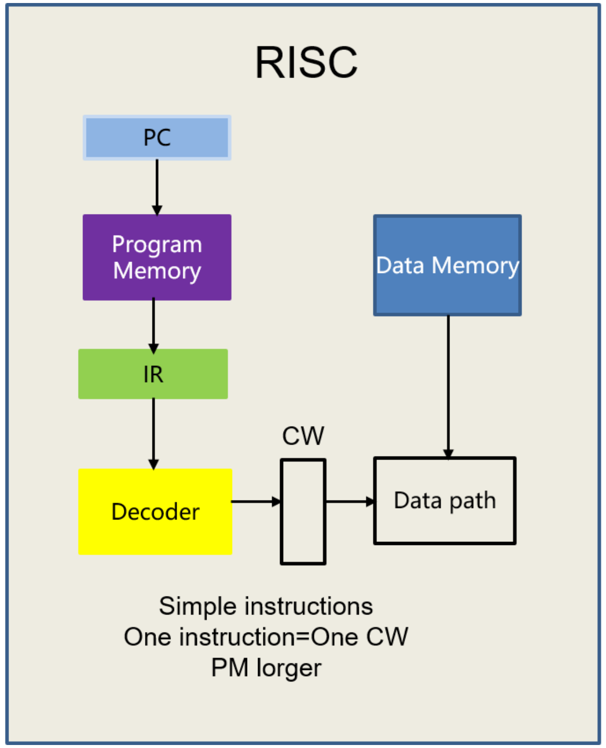
Yasas Sri Wickramasinghe

RISC

- Reduced Instruction Set Computer

- Key features
  - Large number of general purpose registers
  - or use of compiler technology to optimize register use
  - Limited and simple instruction set
    - Normally, one instruction executes one operation
  - All instructions are fix length
  - Instruction execution times are same, constant
  - Control Unit is optimized
  - Emphasis on optimising the instruction pipeline

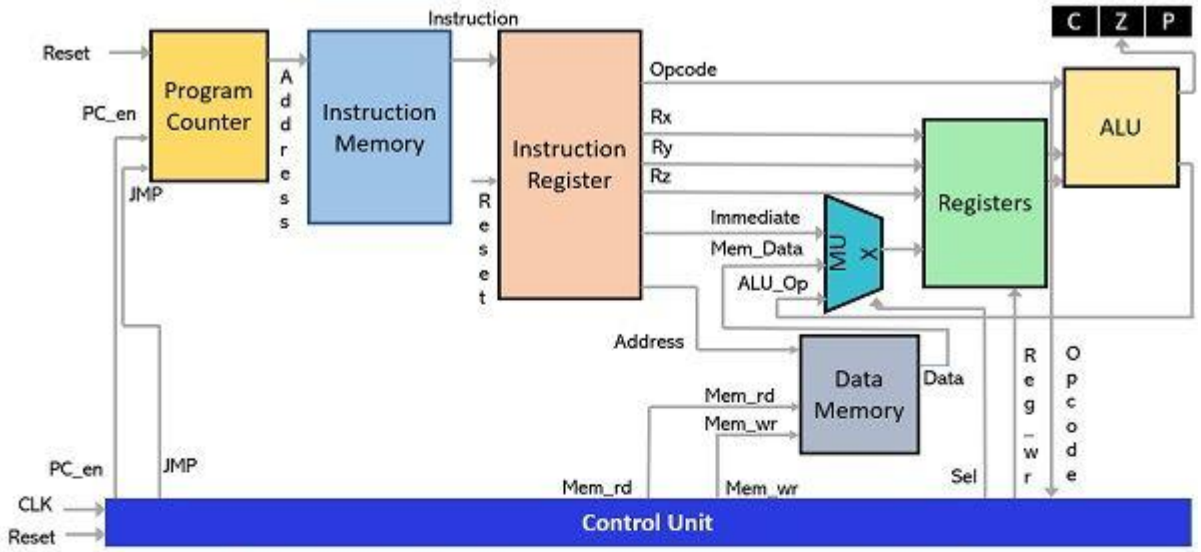




Simple instructions
One instruction=One CW
PM lorger

RISC Characteristics

- One instruction per cycle
- Register to register operations
- Few, simple addressing modes
- Few, simple instruction formats
- Hardwired design (no microcode)
- Fixed instruction format
- More compile time/effort

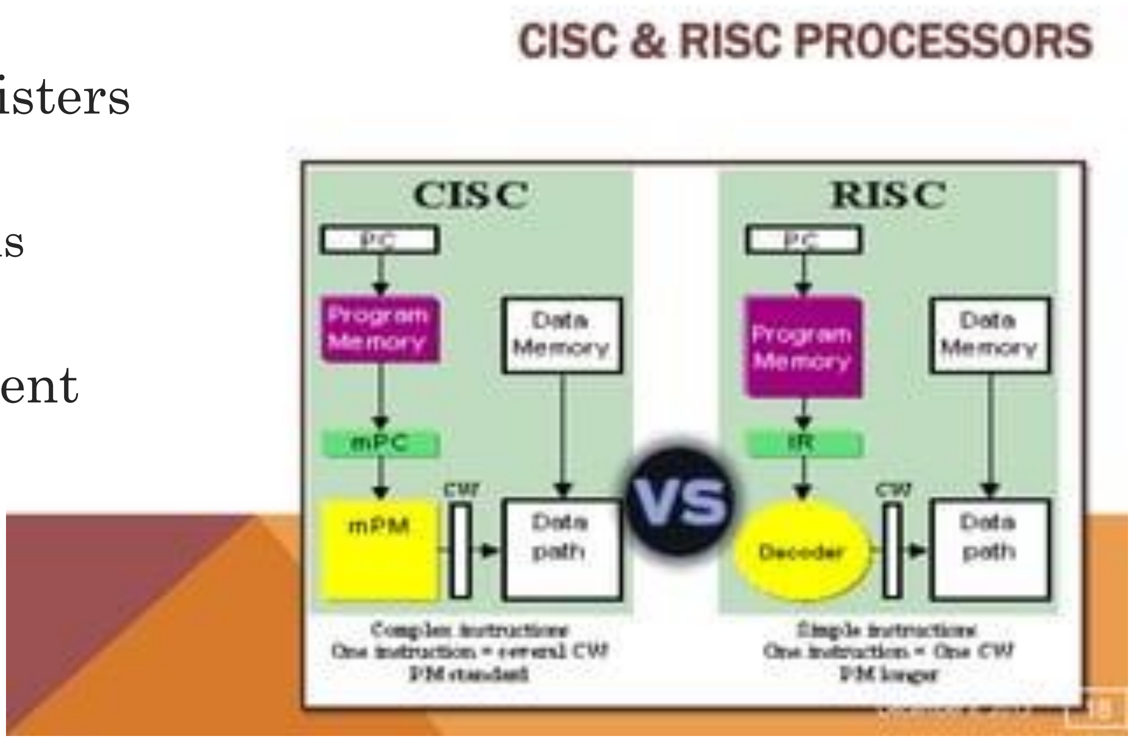


Architecture of RISC

CISC

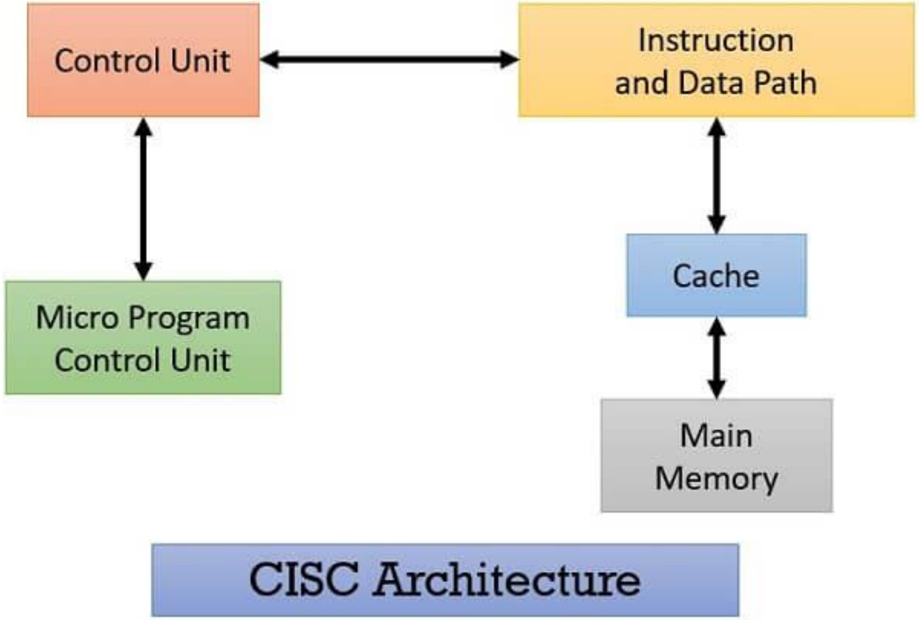
- Complex Instruction Set Computer

- Key features
  - Small number of general purpose registers
  - Complex instructions
    - One instruction can do many operations
  - Variable length instructions
  - Instruction execution times are different
  - Control unit is complex



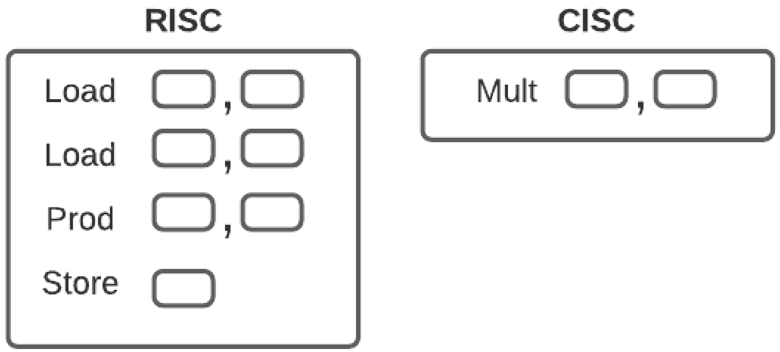
Driving force for CISC

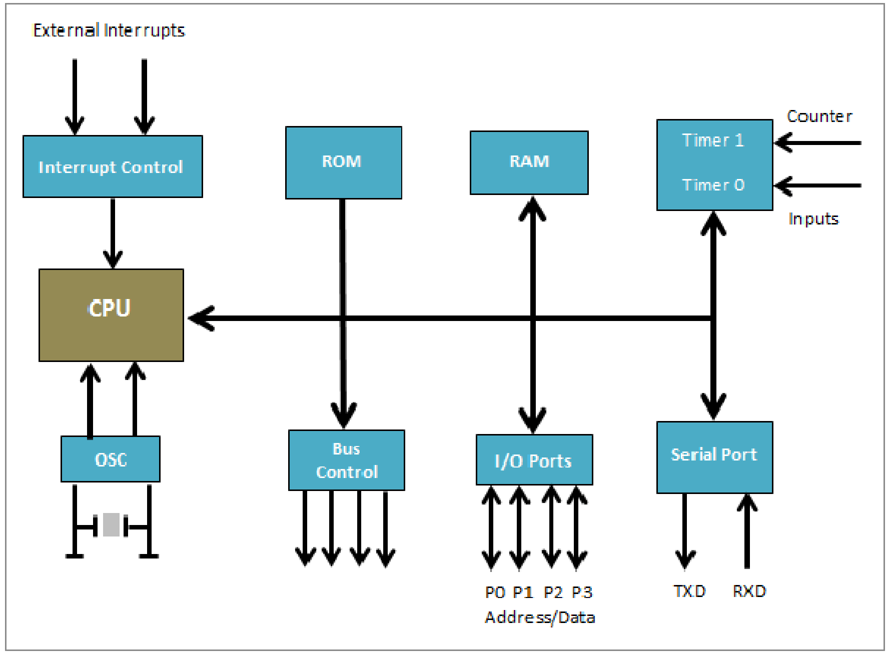
- Software costs far exceed hardware costs
- Increasingly complex high level languages
- Semantic gap
- Leads to:
  - Large instruction sets
  - More addressing modes
  - Hardware implementations of HLL statements
    - e.g. CASE (switch) on VAX
- Ease compiler writing
- Improve execution efficiency
  - Complex operations in microcode
- Support more complex HLLs



Why CISC ?

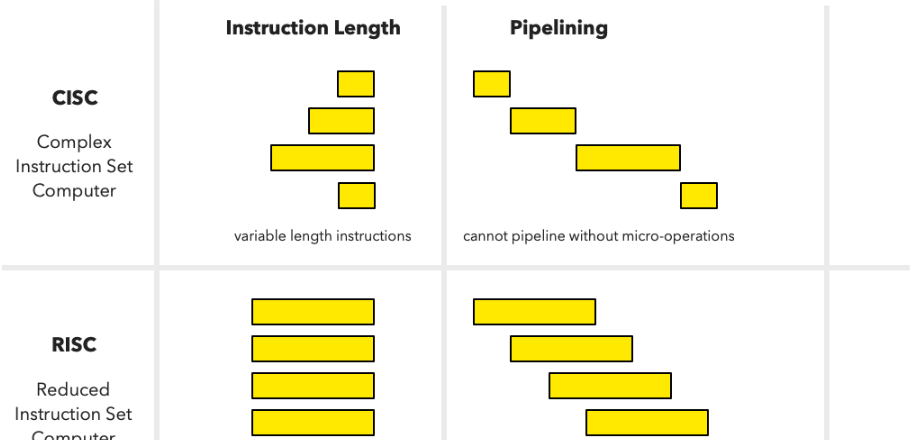
- Compiler simplification?
  - Disputed…
  - Complex machine instructions harder to exploit
  - Optimization more difficult
- Smaller programs?
  - Program takes up less memory but…
  - Memory is now cheap
  - May not occupy less bits, just look shorter in symbolic form
    - More instructions require longer op-codes
    - Register references require fewer bits
- Faster programs?
  - Bias towards use of simpler instructions
  - More complex control unit
  - Microprogram control store larger
  - thus simple instructions take longer to execute
- It is far from clear that CISC is the appropriate solution

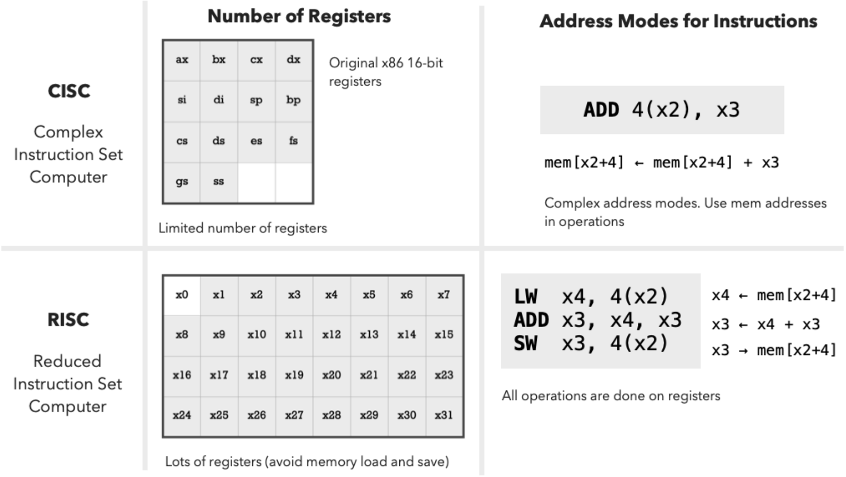




RISC vs CISC

- Not clear cut
- Many designs borrow from both philosophies
- e.g. ARM and x86/x64

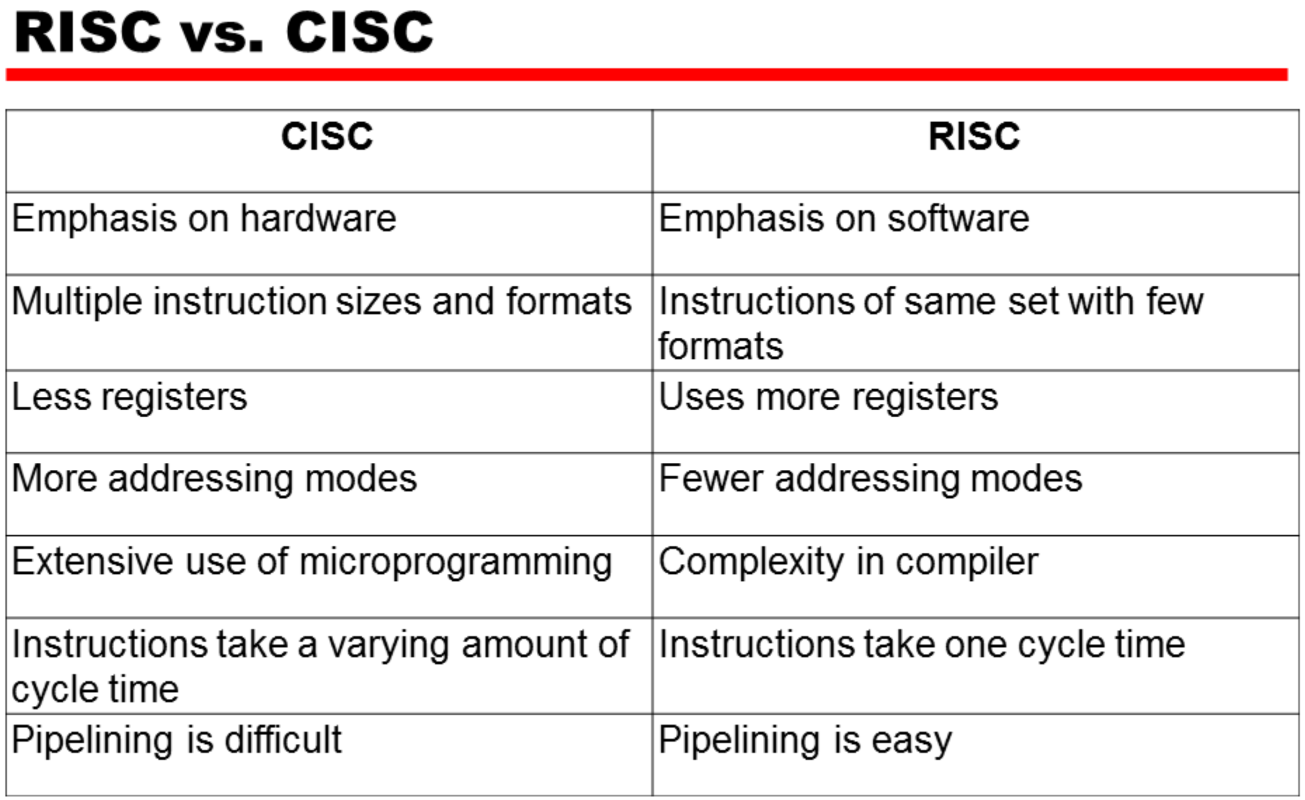

Instruction Length
Pipelining
CISC
Complex Instruction Set Computer
variable length instructions
cannot pipeline without micro-operations
RISC
Reduced Instruction Set Computer


Number of Registers
Address Modes for Instructions
CISC
Complex Instruction Set Computer
ax bx cx dx
si di sp bp
cs ds es fs
gs ss
Original x86 16-bit registers
Limited number of registers
Complex address modes. Use mem addresses in operations
RISC
Reduced Instruction Set Computer
x0 x1 x2 x3 x4 x5 x6 x7
x8 x9 x10 x11 x12 x13 x14 x15
x16 x17 x18 x19 x20 x21 x22 x23
x24 x25 x26 x27 x28 x29 x30 x31
Lots of registers (avoid memory load and save)
All operations are done on registers

```
ADD 4(x2), x3
```

```
mem[x2+4] ← mem[x2+4] + x3
```

```
LW  x4, 4(x2)     x4 ← mem[x2+4]
ADD x3, x4, x3    x3 ← x4 + x3
SW  x3, 4(x2)     x3 → mem[x2+4]
```

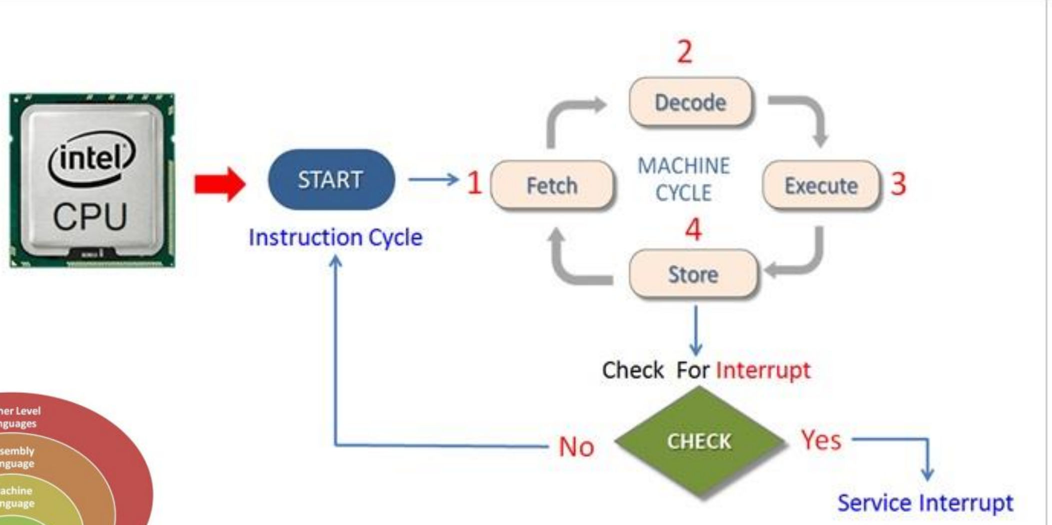


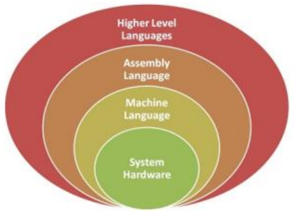
RISC vs. CISC

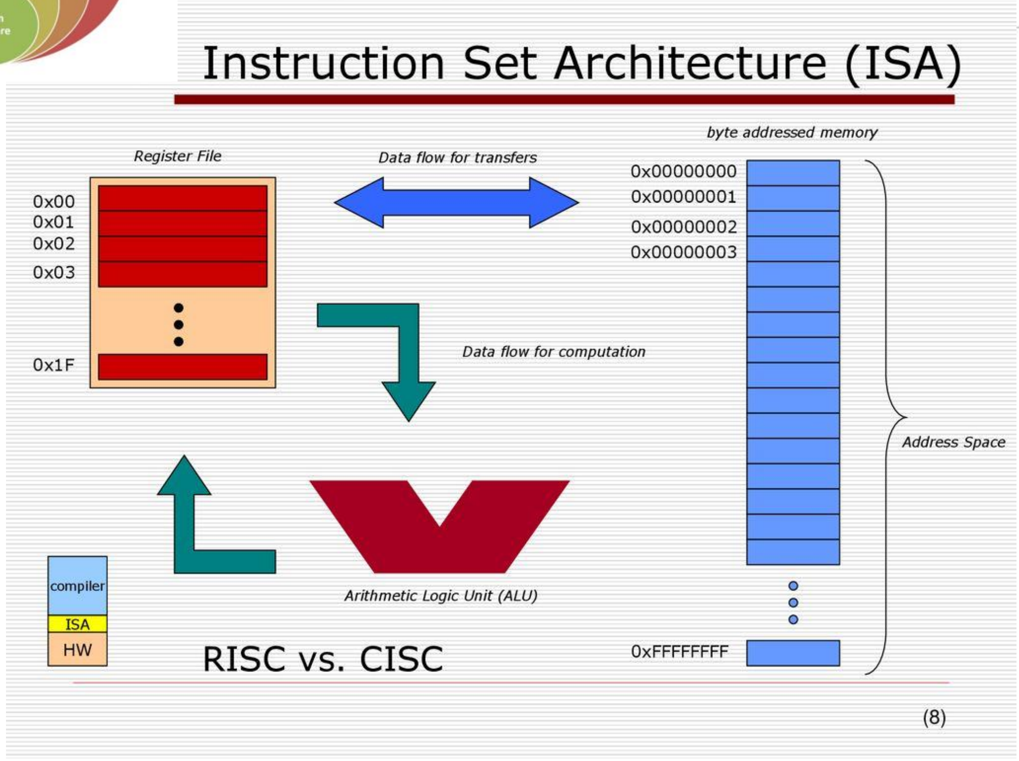
CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy

Conclusion

- Program execution may be interrupted for other tasks
- Instruction Cycle
  - Fetch, Execution, Interrupt
- Instruction Set Architecture (ISA)
  - Addressing Mode – Locate Data
  - Register Model – CPU Store
  - Instruction Format
    - Opcode + Operands
  - Byte-Order – Arrange of numbers
- CISC vs RISC have Pros and Cons







Instruction Set Architecture (ISA)

RISC vs. CISC

(8)

END

Questions?