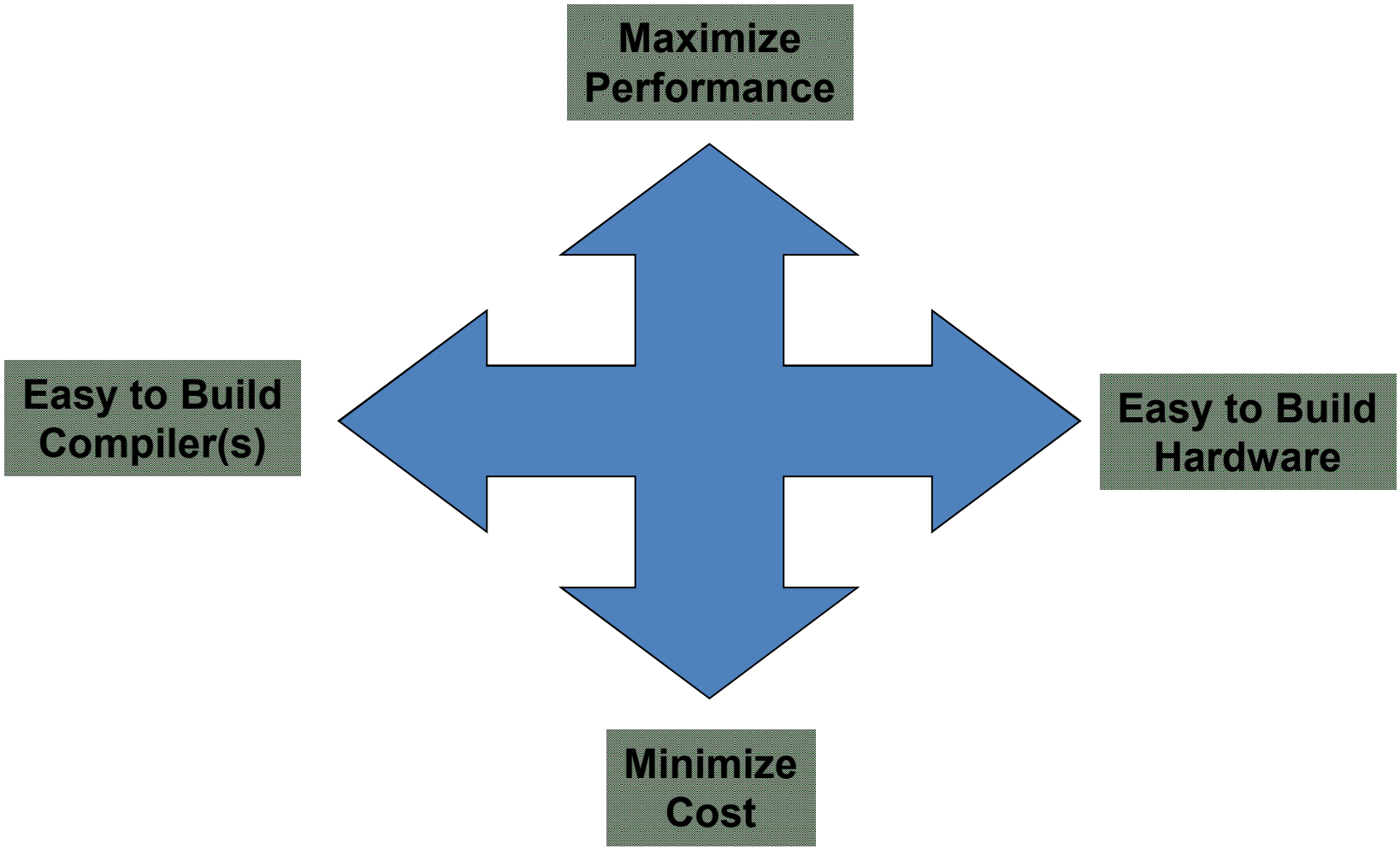
William Stallings
Computer Organization and Architecture
6th Edition

Chapter 10-11
Instruction Sets:
Addressing Modes and Formats

Instruction set

- Instruction set is key differentiation between different processors (e.g. Intel x86 & Power PC)
  - MIPS: All loads and stores 32 bits; Special instructions exist for extracting bytes
  - DEC Alpha: Instructions for loading and storing different sizes
  - Some arch. have predefined values e.g. 0, 1, etc.
  - DEC Vax: Single instruction to load or store all registers
- High-level language constructs shape ISA
  - Expr and assignment stmts
  - Conditional stmts
  - Loops
  - Procedure calls
- Compiling high-level language constructs into efficient code

Instruction Set Design Goals (Genel)



Instruction Set Design (specific)

- One goal is to minimize instruction length
- Another goal (in CISC design) is to maximize flexibility
- Many instructions were designed with compilers in mind
- **Operand** in an instruction is actual value (immediate) or a reference to address (of operand)
- Determining instruction format, how of operand address modes: key of **instruction set design**
- Instruction format: layout fields in the instruction
  - Legth (fixed or variable)
  - Number of bits in opcodes and in each operand
- How addressing mode is determined
  - We want to reference a large range of locations in main memory
  - Employ addressing techniques

Addressing Modes

- Different types of addresses involve tradeoffs(denge) b/w instruction length, addressing flexibility(range), complexity of address calculation
- How **address** of an operand specified (opcode / mode field)
- How bits of an instruction organized to define operand address
- (Most Common) **Addressing Modes**
  - Immediate (actual value)
  - Direct (operand address in address field)
  - Indirect(address field points to a loc that has operand address)
  - Register
  - Register Indirect
  - Displacement forms: reg value + address value (Indexed)
  - Implied (Stack, and few others)

Immediate Addressing

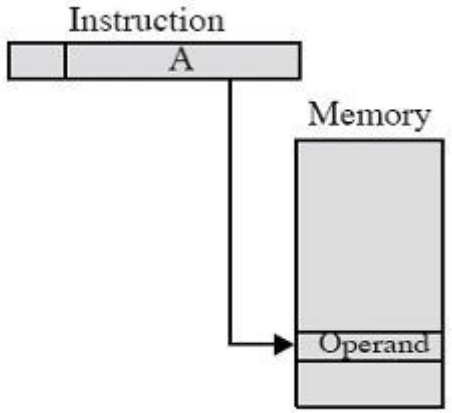
- To set initial values of variables
- Operand value is part of instruction
- Operand = one address field
- e.g. ADD 5
  - —Add 5 to contents of accumulator
  - —5 is operand
- No memory reference to fetch data
- Fast and simple
- Can have limited range in machines with fixed length instructions
- Address calculation

Immediate Addressing and Small Operands

- Many immediate mode instructions use small operands (8 bits)
- In 32 or 64 bit machines with variable length instructions, space is wasted if immediate operands are required to be the same size as the register size
- Some instruction formats include a bit that allows small operands to be used in immediate instructions
- ALU will zero-extend or sign-extend the operand to the register size

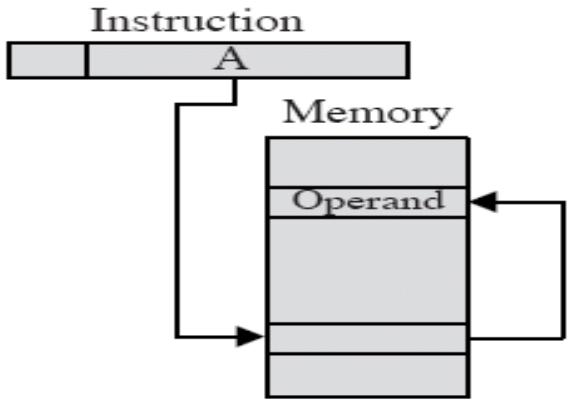
Direct Addressing

- 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



(Memory-) 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
  - —Two memory references to fetch the operand

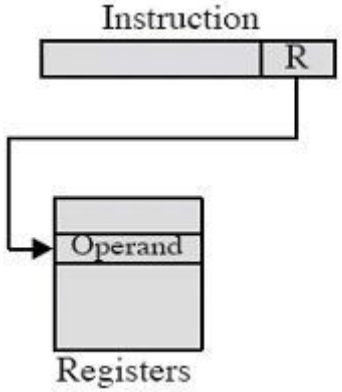


Indirect Addressing (2)

- Large address space; 2^n where n = word length
- May be nested, multilevel, cascaded
  - —e.g. EA = (((A)))
    - Draw the diagram yourself
    - ( ) : contents of
  - —Rarely used;
  - —Multiple memory accesses to find operand
  - —Implemented by using one bit of full-word address as an indirect flag; flag bit I=1 (indirection), I=0 then EA
    - Allows unlimited depth of indirection
- Multiple memory accesses; hence slower than any other type of addressing

Register Addressing (1)

- Similar to direct addressing
- Address field refers to a register (not a memory address). Operand is held in this register.
- EA = R (not contents of R)
- Limited number of general-purpose registers (8-32) exist
  - —Therefore a small address field needed
  - —Shorter instructions
  - —Faster instruction fetch
  - — X86: 3 bits used to specify one of 8 registers



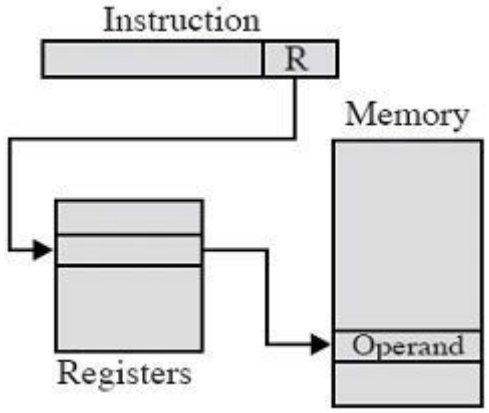
Register Addressing (2)

- No memory access needed to fetch EA
- Very fast execution
- Very limited address space
- Multiple registers can help performance
  - —Requires good assembly programming or compiler writing (use a value in register **many times**)
  - —Note: in C you can specify register variables

    register int a;
  - —This is only advisory to the compiler
  - —No guarantees

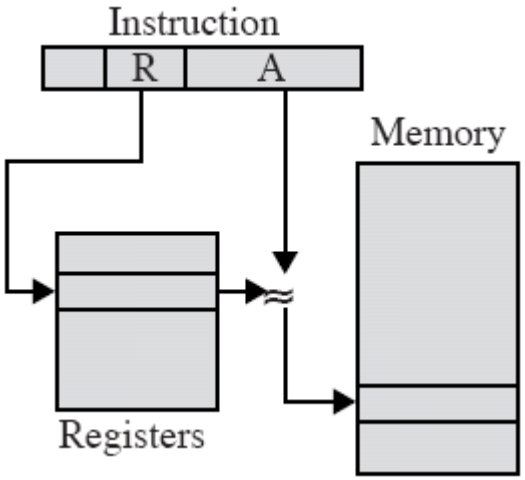
Register Indirect Addressing

- Similar to memory-indirect addressing;
- Much more common
- EA = (R)
- Operand is in memory cell pointed to by contents of reg R
- Large address space (2^n)
- One fewer memory access than indirect addressing



Displacement Addressing

- Very powerful addressing mode
- EA = A + (R)
- Combines reg. indirect addressing with direct addressing
- Address field hold two values, one explicit
  - —A =base value (used directly)
  - —R =register that holds displacement, contents+A->EA
  - —or vice versa



Types of Displacement Addressing

- 3 most common types are
  - —Relative Addressing (implicit R is PC)
  - —Base-register addressing(reg R:mem address, offset)
  - —Indexing (mem address a, R:+offset from a)

Relative Addressing

- A type of displacement addressing
- R = Program counter, PC
- Sometimes called PC-relative addressing
- Address field A treated as 2's complement integer to allow backward references
- EA = A + (PC)
  - —i.e. get operand from A cells from current location pointed to by PC
- Can be very efficient because of locality of reference & cache usage
  - —But in large programs code and data may be widely separated in memory

Base-Register Addressing

- A holds displacement
- R holds pointer to base address
- R may be explicit or implicit
  - —e.g. segment registers in 80x86 are base registers and are involved in all EA computations
- x86 processors have a wide variety of base addressing formats
  - —mov eax, [edi + 4 * ecx]
  - —sub [bx+si-12],2
- Uses locality of memory references
- To implement segmentation

Indexed Addressing

- Opposite usage w.r.t. base-register addressing
  - A = base (memory address)
  - R = displacement (+offset)
  - EA = A + R
- Good for accessing arrays (i.e. A[]) (R called index reg)
  - EA = A + R
  - R++
- Iterative access to sequential memory locations is very common
- Some architectures provide auto-increment or auto-decrement as **auto-indexing**
  - Preindex EA = A + (R++)
  - Postindex EA = A + (++R)

Stack Addressing

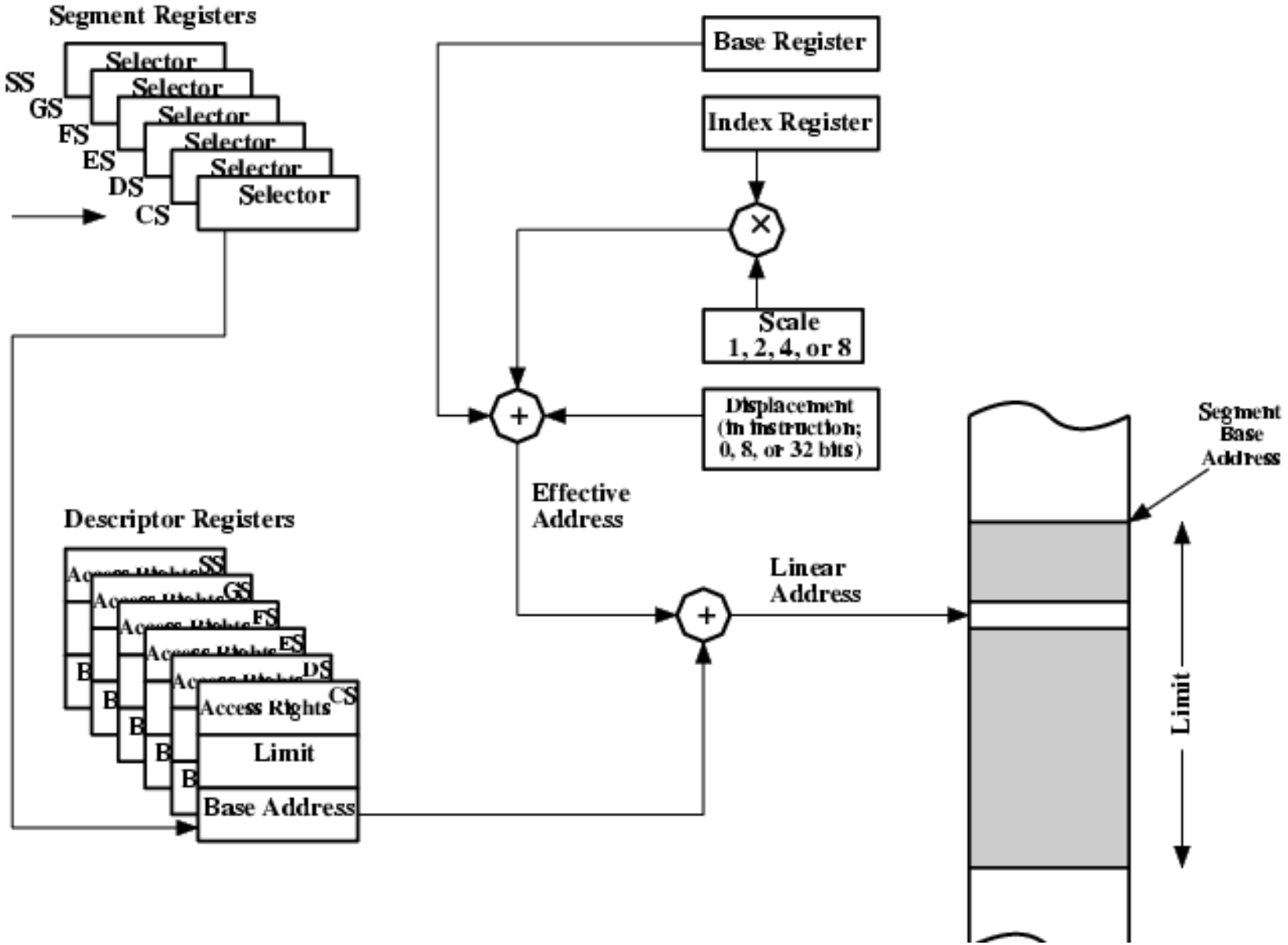
- Stack: a linear array of locations
- top of the stack pointer is associated (in register)
- Operand is (implicitly) on top of stack
- e.g.
  - ADD Pop top two items from stack and add
  - PUSH
  - POP
- X87 is a stack machine so it has instructions such as
- FADDP ; st(1)<- st(1) + st(0) ;pop; result left in st(0)

Örnekler:

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
  - 6 Segment registers: subject of reference
- 12 addressing modes available
  - Immediate (operand in instr)
  - Register operand (operand in register)
  - Displacement
  - Base
  - Base with displacement
  - Scaled index with displacement
  - Base with index and displacement
  - Base scaled index with displacement
  - Relative

Pentium Addressing Mode Calculation

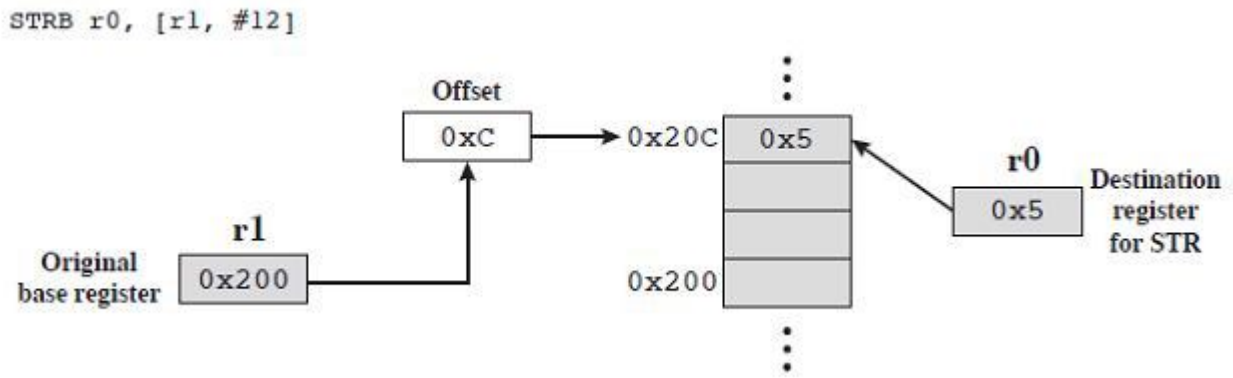


ARM Addressing Modes

- A typical RISC characteristic is a small and simple set of addressing modes
- ARM departs somewhat from this convention with a relatively rich set of addressing modes
- But Load and Store are the only instructions that can reference memory
  - Always indirect through a base register plus offset
- Three alternatives
  - Offset
  - Preindex
  - Postindex

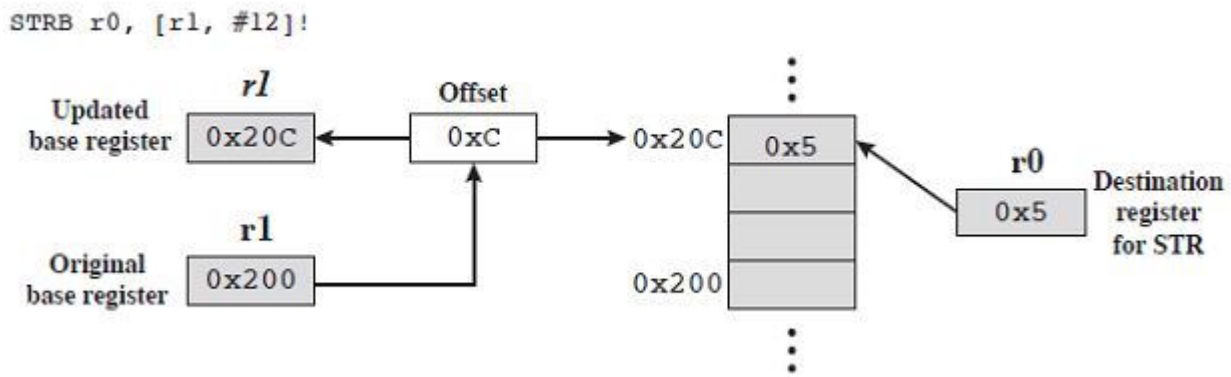
Offset Addressing

- Offset added or subtracted from value in base register
- Example: store byte, base register is R1 and displacement is decimal 12. This is the address where the byte from r0 is stored



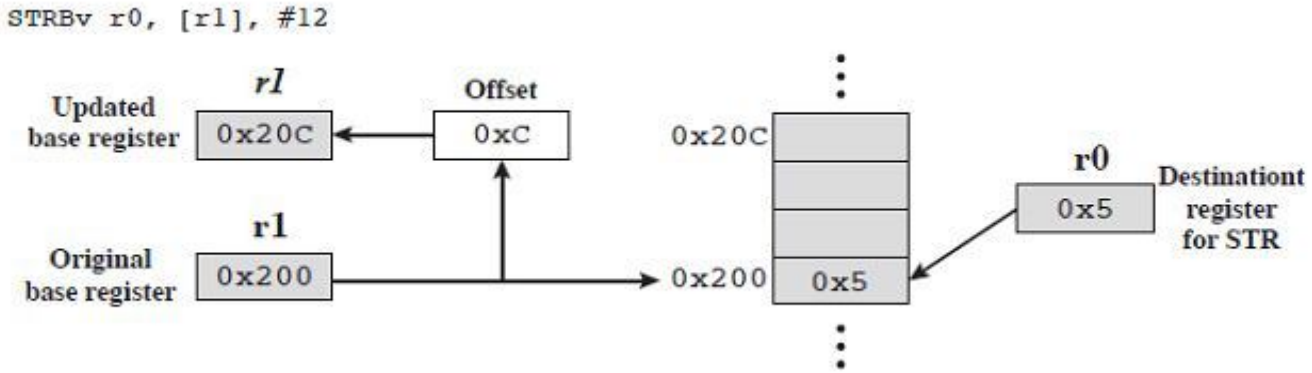
Preindex Addressing

- Memory address formed same way as offset addressing, but the memory address is written back to the base register after adding or subtracting the displacement
- With preindexing the writeback occurs before the store to memory



Postindex Addressing

- Like preindex addressing but the writeback of the effective address occurs after the store



Indexed Addressing Operands

- Previous examples had immediate values but the offset or displacement can also be in another register
- If a register is used then addresses can be scaled
- The value in the offset register is scaled by one of the shift operators
  - Logical Shift Left / Right
  - Arithmetic Shift Right
  - Rotate Right
  - Rotate Right Extended
  - Amount of shift is an immediate operand in the instruction

Arithmetic and Logical Instructions

- Use register and immediate operands only
- For register addressing one of the register operands can be scaled by one of the five shift operations mentioned previously
- **Branch Instructions**
- Only form of addressing is immediate
- Branch instruction contains a 24-bit immediate value
- For address calculation this value is shifted left by 2 bits so the address is on a word boundary
- This provides a range of 26 bits so we have backwards or forwards branches of 32MB

Load/Store Multiple

- Load/Store multiple loads or stores a subset of general purpose registers (possibly all) from/to memory
- List of registers is specified in a 16-bit field in the instruction (one bit/register)
- Memory addresses are sequential; low address has lowest numbered register
- Found addressing modes:
- — Increment/decrement before/after
- — Base reg specifies a main memory address
- — Inc/Dec starts before/after the first memory access

Instruction Formats

- Defines Layout of bits in an instruction
- Includes opcode
- Includes (implicit or explicit) operand(s)
- Usually more than one instruction format in an instruction set
- Includes implicit or explicit operand(s)
- Usually there are several instruction formats in an instruction set
- Huge variety of instruction formats have been designed; they vary widely from processor to processor

Instruction Length

- The most basic issue
- Affected by and affects:
  - Memory size
  - Memory organization
  - Bus structure
  - CPU complexity
  - CPU speed
- Trade off between a powerful instruction repertoire and saving space with shorter instructions

Instruction format trade-offs

- Large instruction set => small programs
- Small instruction set => large programs
- Large memory => longer instructions
- Fixed length instructions same size or multiple of bus width => fast fetch
- Variable length instructions may need extra bus cycles
- Processor may execute faster than fetch
- Use cache memory or use shorter instructions
- Complex relationship between word size, character size, instruction size and bus transfer width
- In almost all modern computers these are all multiples of 8 and related to each other by powers of 2

Allocation of Bits: Determines several important factors

- Number of addressing modes
  - — Implicit operands don't need bits
  - — X86 uses 2-bit mode field to specify interpretation of 3-bit operand fields
- Number of operands
  - — 3 operand formats are rare,
  - — For two operand instructions we can use one or two operand mode indicators
- Register versus memory
  - —Tradeoff between # of registers and program size
  - — Studies suggest optimal number between 8 and 32
  - —Most newer architectures have 32 or more

Allocation of Bits

- Number of register sets
  - RISC architectures tend to have larger sets of uniform registers
  - Small register sets require fewer opcode bits
  - Specialized register sets can reduce opcode bits further by implicit reference (address vs. data registers)
- Address range
  - Large address space requires large instructions for direct addressing
  - Many architectures have some restricted or short forms of displacement addressing

PDP-8

- Very simple machine and instruction set
- Has one register (the Accumulator)
- 12-bit instructions operate on 12-bit words
- Very efficient implementation
  - —35 operations along with indirect addressing, displacement addressing and indexing in 12 bits
- The lack of registers is handled by using part of the first physical page of memory as a register file

Instruction Formats

- A 3-bit opcode and three types of instructions
  - —For opcodes 0 – 5 (6 basic instructions) we have single address mem ref with Z/C I/D bits
- Opcode 6 is I/O with 6 device-select bits and 3 operation bits
- Opcode 7 defines a register reference or microinstruction
  - — Three groups, where bits are used to specify operation (e.g., clear accumulator)
  - — Forerunner of modern microprogramming

PDP-8 Instruction Format

Memory Reference Instructions

Opcode			D/I	Z/C	Displacement						
0	1	2	3	4	5	6	7	8	9	10	11

Input/Output Instructions

1	1	0	Device						Opcode		
0	1	2	3	4	5	6	7	8	9	10	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

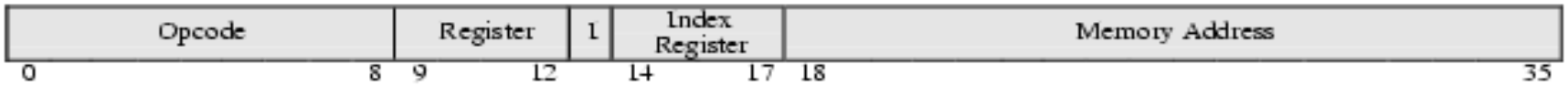
Group 3 Microinstructions

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 Accumultator Right
RAL = Rotate Accumulator Left
BSW = Byte SWap

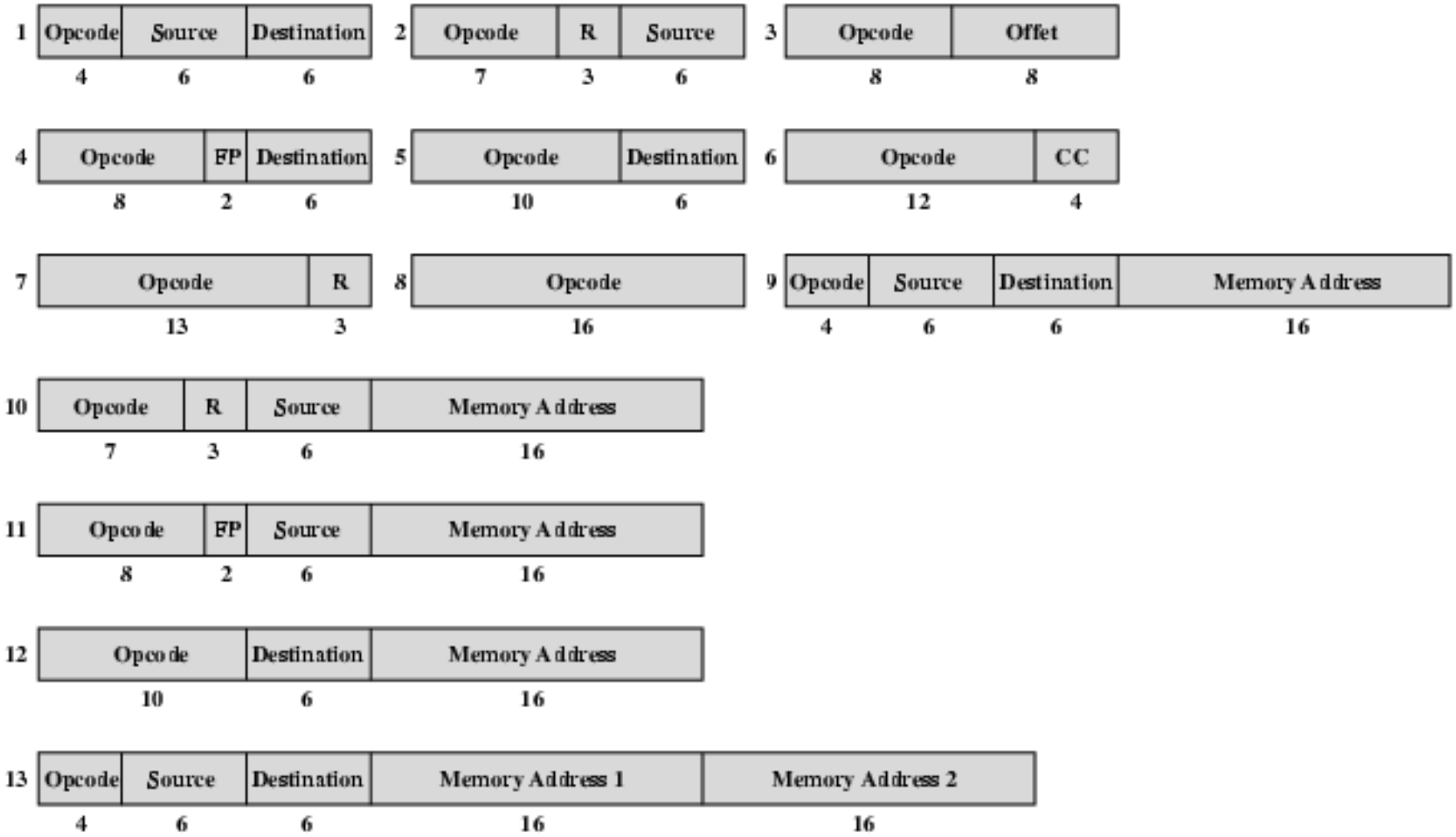
IAC = Increment ACcumulator
SMA = Skip on Minus Accumulator
SZA = Skip on Zero Accumulator
SNL = Skip on Nonzero Link
RSS = Reverse Skip Sense
OSR = Or with Switch Register
HLT = HaLT
MQA = Multiplier Quotient into Accumulator
MQL = Multiplier Quotient Load

PDP-10 Instruction Format



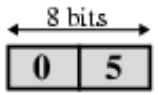
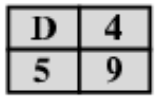
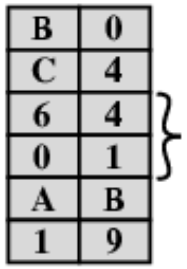
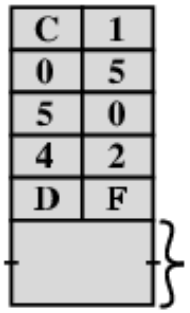
I = indirect bit

PDP-11 Instruction Format

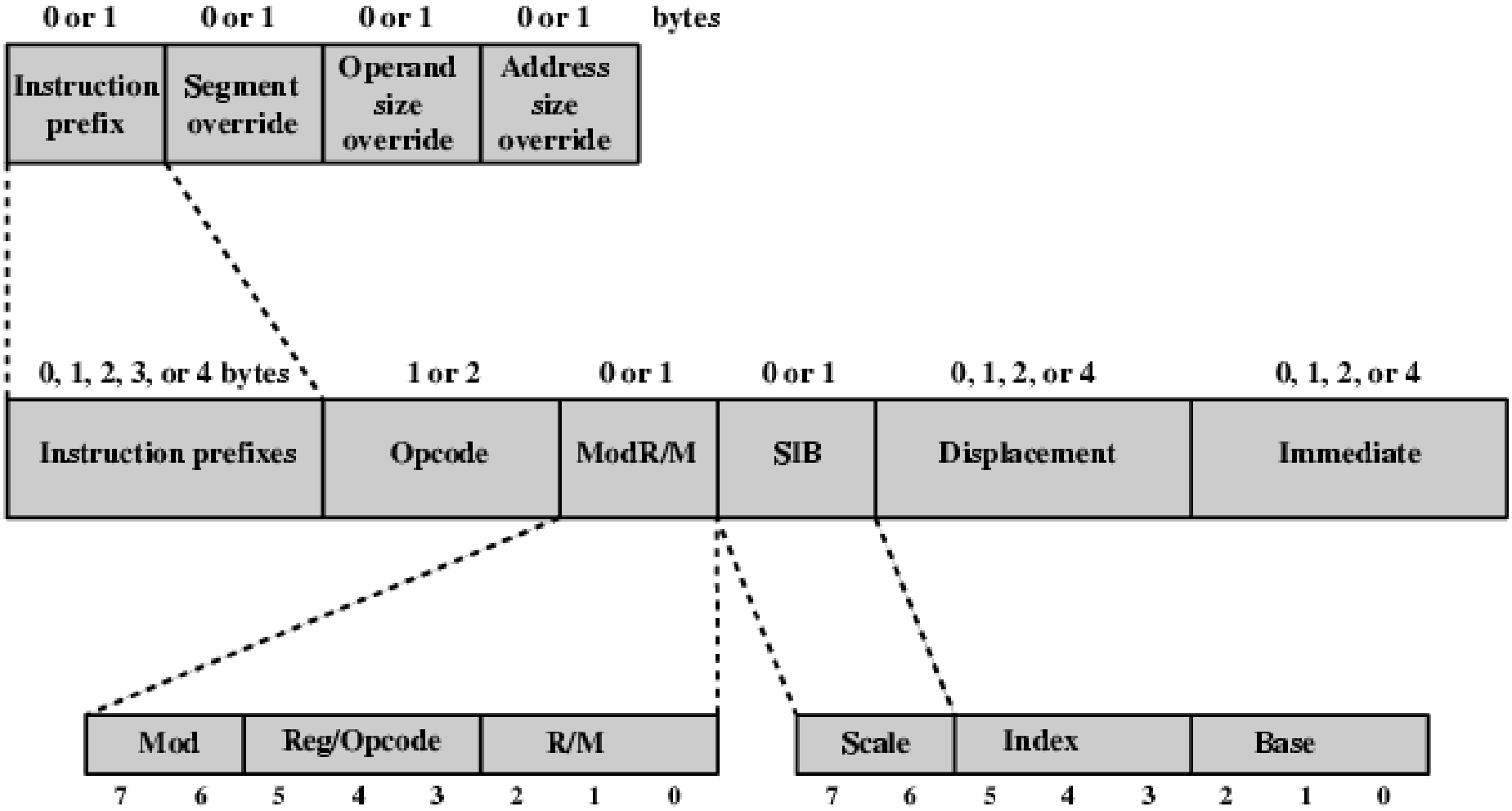


Numbers below fields indicate bit length
Source and Destination each contain a 3-bit addressing mode field and a 3-bit register number
FP indicates one of four floating-point registers
R indicates one of the general-purpose registers
CC is the condition code field

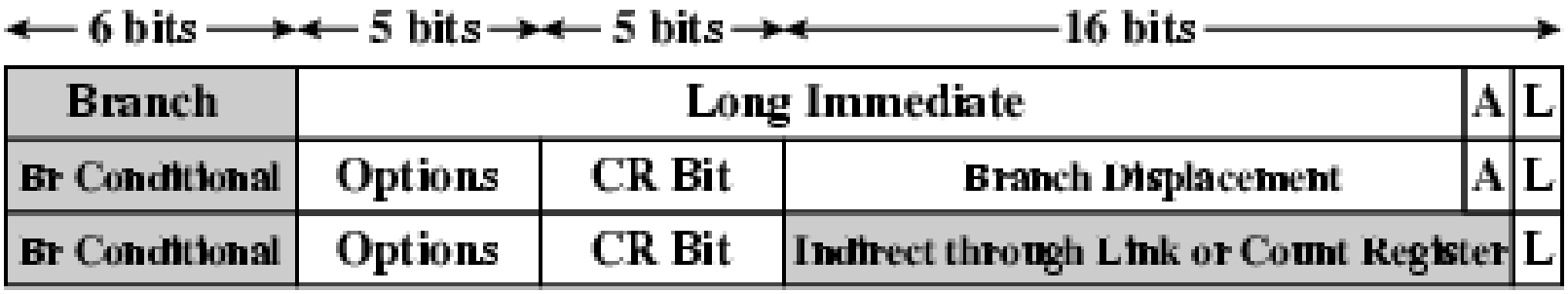
VAX Instruction Examples

Hexadecimal Format	Explanation	Assembler Notation and Description
	Opcode for RSB	RSB Return from subroutine
	Opcode for CLRL Register R9	CLRL R9 Clear register R9
	Opcode for MOVW Word displacement mode, Register R4 356 in hexadecimal Byte displacement mode, Register R11 25 in hexadecimal	MOVW 356(R4), 25(R11) Move a word from address that is 356 plus contents of R4 to address that is 25 plus contents of R11
	Opcode for ADDL3 Short literal 5 Register mode R0 Index prefix R2 Indirect word relative (displacement from PC) Amount of displacement from PC relative to location A	ADDL3 #5, R0, @A[R2] Add 5 to a 32-bit integer in R0 and store the result in location whose address is sum of A and 4 times the contents of R2

Pentium Instruction Format

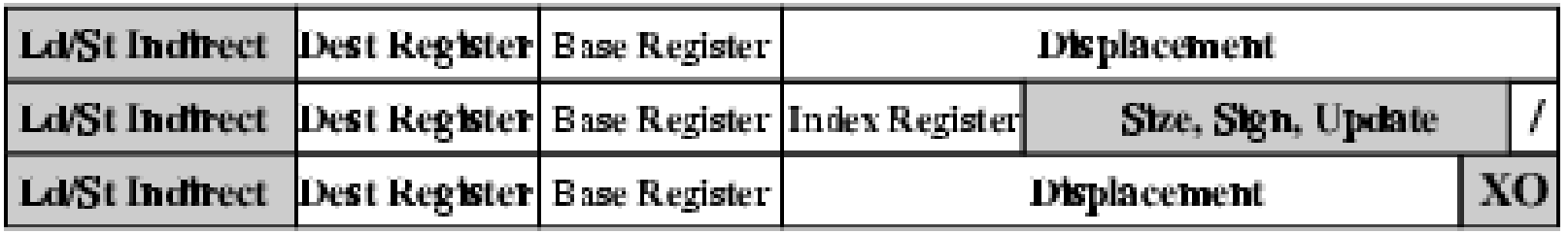


PowerPC Instruction Formats (1)



(a) Branch instructions

(b) Condition register logical instructions



(c) Load/store instructions

PowerPC Instruction Formats (2)

Ld/St Indirect	Dest Register	Base Register	Displacement				
Ld/St Indirect	Dest Register	Base Register	Index Register	Size, Sign, Update	/		
Ld/St Indirect	Dest Register	Base Register	Displacement		XO		*

(c) Load/store instructions

Arithmetic	Dest Register	Src Register	Src Register	O	Add, Sub, etc.		R	
Add, Sub, etc.	Dest Register	Src Register	Signed Immediate Value					
Logical	Src Register	Dest Register	Src Register	ADD, OR, XOR, etc.			R	
AND, OR, etc.	Src Register	Dest Register	Unsigned Immediate Value					
Rotate	Src Register	Dest Register	Shift Amt	Mask Begin	Mask End		R	
Rotate or Shift	Src Register	Dest Register	Src Register	Shift Type or Mask			R	
Rotate	Src Register	Dest Register	Shift Amt	Mask	XO	S	R	*
Rotate	Src Register	Dest Register	Src Register	Mask	XO		R	*
Shift	Src Register	Dest Register	Shift Type or Mask			S	R	*

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

Flt sgl/dbl	Dest Register	Src Register	Src Register	Src Register	Fadd, etc.	R

(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

VAX Design Philosophy

- Most architectures provide a fairly small number of fixed instruction formats
  - —Addressing mode and opcode are not orthogonal
  - —Instructions typically limited to reg/mem, reg/reg etc
  - —Only 2 or 3 operands max can be accommodated; some instructions inherently require more (ex: integer division with 2 inputs and 2 outputs)
- VAX design principles were:
  - —All instructions should have the "natural" number of operands
  - —All operands should have the same generality in specification

VAX Instructions

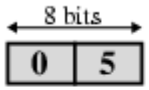
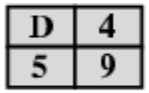
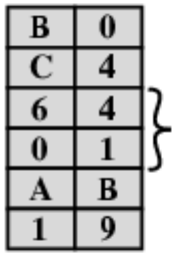
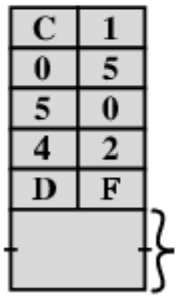
- Highly variable instruction format
- Opcode is one or two bytes long
  - First by FF or FD indicates two byte opcode
- Followed by 0 to 6 operand specifiers
- Minimum instruction length is one byte
- Maximum is 37 bytes!

Operand Specifiers

- At a minimum, 1 byte in which leftmost 4 bits are the address mode specifier
  - —except "literal" mode (00 followed by 6 literal bits)
  - —4 bits specify one of 16 registers
- Operand specifier can be extended by immediate or displacement 8 – 32 bits
- Indexed addressing mode 0010 + 4 bit index reg id, followed by base address 8-32 bits
- **Example: 6 operand instruction**

  ADDP6 OP1, OP2, OP3, OP4, OP5, OP6
  - —Adds two packed decimal strings
  - —Op1 and op2 are length and start addr of one string; op3 and 4 are second string- Result is in length/location op5,op6

VAX Instruction Examples

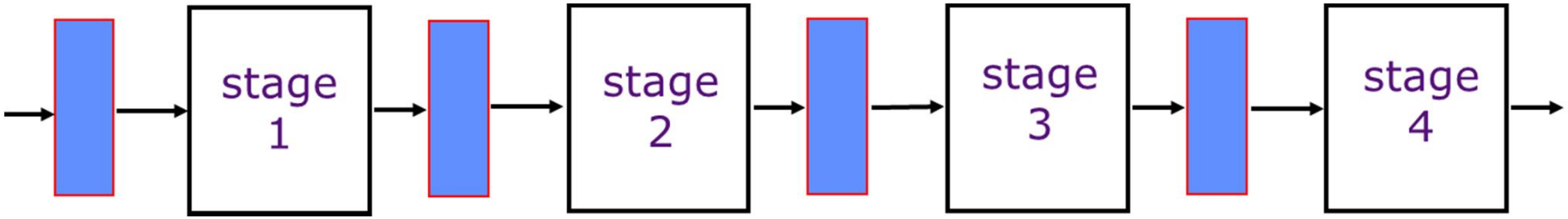
Hexadecimal Format	Explanation	Assembler Notation and Description
	Opcode for RSB	RSB Return from subroutine
	Opcode for CLRL Register R9	CLRL R9 Clear register R9
	Opcode for MOVW Word displacement mode, Register R4 356 in hexadecimal Byte displacement mode, Register R11 25 in hexadecimal	MOVW 356(R4), 25(R11) Move a word from address that is 356 plus contents of R4 to address that is 25 plus contents of R11
	Opcode for ADDL3 Short literal 5 Register mode R0 Index prefix R2 Indirect word relative (displacement from PC) Amount of displacement from PC relative to location A	ADDL3 #5, R0, @A[R2] Add 5 to a 32-bit integer in R0 and store the result in location whose address is sum of A and 4 times the contents of R2

Pipelining

- In pipelining we divide instruction execution into a number of stages: Overlap these operations
  - Fetch instruction
  - Decode instruction
  - Calculate operands (i.e. EAs)
  - Fetch operands
  - Execute instructions
  - Write result
- After one stage has completed, instruction moves down the pipeline to the next stage while the next instruction is started
  - Similar to a factory assembly line – we don't have to wait for a product to exit the line before starting to assemble another

An Ideal Pipeline

- Complex instruction sets difficult to pipeline, so difficult to increase performance as gate count grew
- Load-Store RISC ISAs designed for efficient pipelined implementations



- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

Sample Pipeline

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

Instruction-Fetch (IF)

Decode, Reg. Fetch (ID)

Execute (EX)

Memory (MA)

Write-Back (WB)

5 stages = 5 instr execution at the same time (see t4)

Instructions interact with each other in pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*
- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value
    → *data hazard*
  - Dependence may be for the next instruction's address
    → *control hazard (branches, exceptions)*

Foreground Reading

- Processor examples
- Stallings Chapter 12
- Web pages etc.