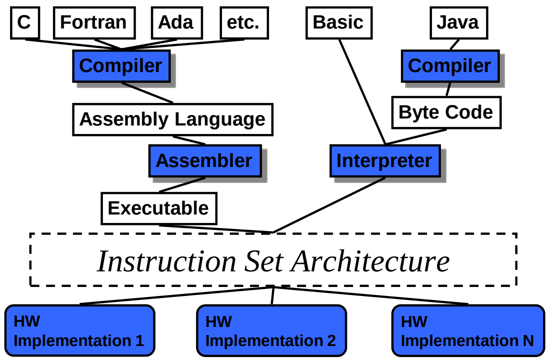
BLM6112
Advanced Computer Architecture
Instruction Set Architecture

Prof. Dr. Nizamettin AYDIN

naydin@yildiz.edu.tr

http://www3.yildiz.edu.tr/~naydin



Instruction Set

- Instruction: Language of the machine
- Instruction set: Vocabulary of the language (Collection of instructions that are understood by a CPU)
  lda, sta, brp, jmp, nop, …
- Machine Code
  - machine readable
  - Binary(example: 1000110010100000)
- Usually represented by assembly codes
  - Human readable
    - Example: adding a number entered from keyboard and a number in memory location 40

```
0    in
1    sta    30
2    add    40
3    sta    50
4    hlt
```

Instruction Types

- Data processing
  - ADD, SUB
- Data storage (main memory)
  - STA
- Data movement (I/O)
  - IN, OUT, LDA
- Program flow control
  - BRZ

Elements of an Instruction

- Operation code (Op-code)
  - Do this
    - Example: ADD 30
- Source Operand reference
  - To this
    - Example: LDA 50
- Result Operand reference
  - Put the result here
    - Example: STA 60
- Next Instruction Reference
  - When you have done that, do this...
    - PC points to the next instruction

Source and Result Operands

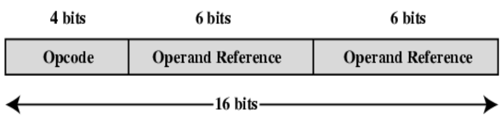
- Source and Result Operands can be in one of the following areas:
  - Main memory
  - Virtual memory
  - Cache
  - CPU register
  - I/O device

Instruction Representation

- In machine code each instruction has a unique bit pattern
- For human consumption a symbolic representation is used (assembly language)
- Opcodes are represented by abbreviations, called mnemonics indicating the operation
  - ADD, SUB, LDA, BRP, ...
- In an assembly language, operands can also be represented as following
  - ADD A,B (add contents of B and A and save the result into A)

Simple Instruction Format

- Following is a 16 bit instruction format



- So...
  - What is the maximum number of instructions in this processor?
  - What is the maximum directly addressable memory size?

Instruction Set Classification

- One way for classification:
  - Number of operands for typical arithmetic instruction

```
add $s1, $s2, $s3
```

$\rightarrow$ **3**

  - What are the possibilities?
  - Will use this C statement as an example:

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

  - Assume **a**, **b** and **c** are in memory

Zero Address Machine

- a.k.a. Stack Machines
- Example: a = b + c;

```
PUSH b      # Push b onto stack
PUSH c      # Push c onto stack
ADD         # Add top two items
            # on stack and replace
            # with sum
POP  a      # Remove top of stack
            # and store in a
```

One Address Machine

- a.k.a. Accumulator Machine
- One operand is implicitly the accumulator
- Example: a = b + c;

```
LOAD  b     # ACC ← b
ADD   c     # ACC ← ACC + c
STORE a     # a   ← ACC
```

Two Address Machine (1)

- a.k.a. Register-Memory Instruction Set
- One operand may be a value from memory
- Machine has **n** general purpose registers
  - $0 through $n-1
- Example: a = b + c;

```
LOAD  $1, b   # $1   ← M[b]
ADD   $1, c   # $1   ← $1 + M[c]
STORE $1, a   # M[a] ← $1
```

Two Address Machine (2)

- a.k.a. Memory-Memory Machine
- Another possibility do stuff in memory!
- These machines have registers used to compute memory addresses
- 2 addresses (One address doubles as operand and result)
- Example: a = b + c;

```
MOVE  a, b   # M[a] ← M[b]
ADD   a, c   # M[a] ← M[a] + M[c]
```

Two Address Machine (3)

- a.k.a. Load-Store Instruction Set or Register-Register Instruction Set
- Typically can only access memory using load/store instructions
- Example: a = b + c;

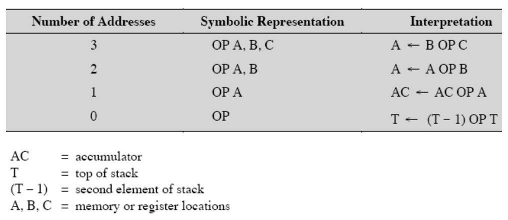
```
LOAD  $1, b  # $1 ← M[b]
LOAD  $2, c  # $2 ← M[c]
ADD   $1, $2 # $1 ← $1 + $2
STORE $1, a  # M[a] ← $1
```

Three Address Machine

- a.k.a. Load-Store Instruction Set or Register-Register Instruction Set
- Typically can only access memory using load/store instructions
- 3 addresses (Operand 1, Operand 2, Result)
  - May be a forth - next instruction (usually implicit)
  - Needs very long words to hold everything
- Example: a = b + c;

```
LOAD  $1, b       # $1 ← M[b]
LOAD  $2, c       # $2 ← M[c]
ADD   $3, $1, $2  # $3 ← $1 + $2
STORE $3, a       # M[a] ← $3
```

Utilization of Instruction Addresses



Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	A ← B OP C
2	OP A, B	A ← A OP B
1	OP A	AC ← AC OP A
0	OP	T ← (T − 1) OP T

AC = accumulator
T = top of stack
(T − 1) = second element of stack
A, B, C = memory or register locations

Types of Operand

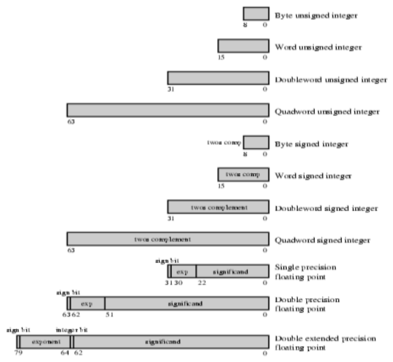
- Addresses
  - Operand is in the address
- Numbers (actual operand)
  - Integer or fixed point
  - floating point
  - decimal
- Characters (actual operand)
  - ASCII etc.
- Logical Data (actual operand)
  - Bits or flags

Pentium Data Types

- 8 bit (byte), 16 bit (word), 32 bit (double word), 64 bit (quad word)
- Addressing in Pentium is by 8 bit units
- A 32 bit double word is read at addresses divisible by 4:

0100	1A	22	F1	77
	+0	+1	+2	+3

Pentium Numeric Data Formats



PowerPC Data Types

- 8 (byte), 16 (halfword), 32 (word) and 64 (doubleword) length data types
- Fixed point processor recognises:
  - Unsigned byte, unsigned halfword, signed halfword, unsigned word, signed word, unsigned doubleword, byte string (<128 bytes)
- Floating point
  - IEEE 754
  - Single or double precision

Types of Operation

- Data Transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System Control
- Transfer of Control

Data Transfer

- Need to specify
  - Source
  - Destination
  - Amount of data
- May be different instructions for different movements
- Or one instruction and different addresses

Arithmetic

- Basic arithmetic operations are...
  - Add
  - Subtract
  - Multiply
  - Divide
  - Increment (a++)
  - Decrement (a--)
  - Negate (-a)
  - Absolute
- Arithmetic operations are provided for...
  - Signed Integer
  - Floating point?
  - Packed decimal numbers?

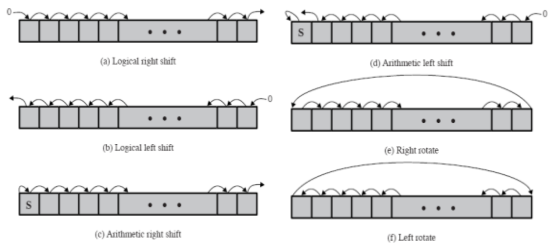
Logical

- Bitwise operations
- AND, OR, NOT
  - Example1: bit masking using AND operation
    - (R1) = 10100101
    - (R2) = 00001111
    - (R1) AND (R2) = 00000101
  - Example2: taking ones coplement using XOR operation
    - (R1) = 10100101
    - (R2) = 11111111
    - (R1) XOR (R2) = 01011010

Basic Logical Operations

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P=Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1

Shift and Rotate Operations



Examples of Shift and Rotate Operations

Input	Operation	Result
10100110	Logical right shift (3 bits)	00010100
10100110	Logical left shift (3 bits)	00110000
10100110	Arithmetic right shift (3 bits)	11110100
10100110	Arithmetic left shift (3 bits)	10110000
10100110	Right rotate (3 bits)	11010100
10100110	Left rotate (3 bits)	00110101

An example - sending two characters in a word

- Suppose we wish to transmit characters of data to an I/O device, 1 character at a time.
  - If each memory word is 16 bits in length and contains two characters, we must unpack the characters before they can be sent.
- To send the left-hand character:
  - Load the word into a register
  - AND with the value 1111111100000000
    - This masks out the character on the right

An example - sending two characters in a word

  - Shift to the right eight times
    - This shifts the remaining character to the right half of the register
  - Perform I/O
    - The I/O module reads the lower-order 8 bits from the data bus.
- To send the right-hand character:
  - Load the word again into the register
  - AND with 0000000011111111
  - Perform I/O

Conversion

- Conversion instructions are those that change the format or operate on the format of data.
- For example:
  - Binary to Decimal conversion

Input/Output

- May be specific instructions
  - IN, OUT
- May be done using data movement instructions (memory mapped)
- May be done by a separate controller (DMA)

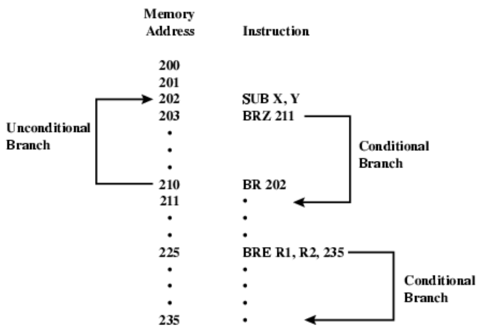
Systems Control

- Privileged instructions
- CPU needs to be in specific state
- For operating systems use

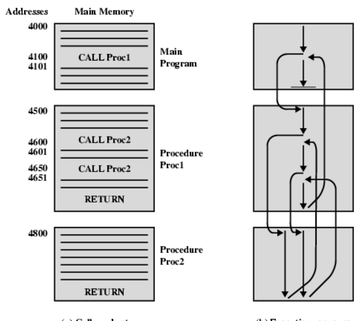
Transfer of Control

- Branch
  - For example: brz 10 (branch to 10 if result is zero)
- Skip
  - e.g. increment and skip if zero
- Subroutine call
  - c.f. interrupt call

Branch Instruction

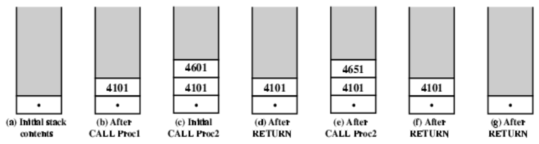


Nested Procedure Calls

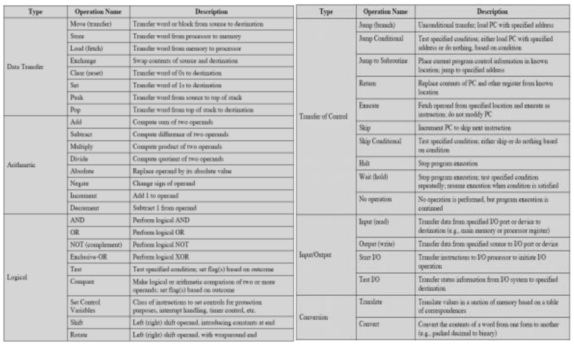


(a) Calls and returns (b) Execution sequence

Use of Stack



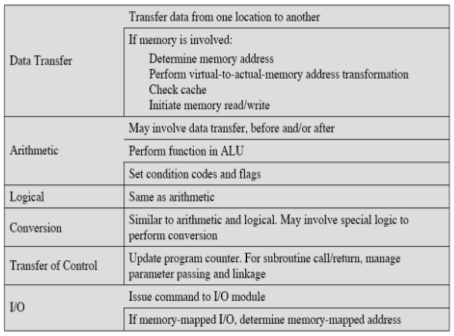
Types of Operation



Type	Operation Name	Description
Data Transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand
Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT (complement)	Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end

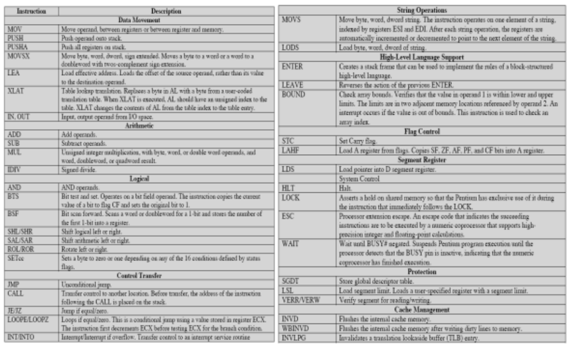
Type	Operation Name	Description
Transfer of Control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued
Input/Output	Input (read)	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O	Transfer status information from I/O system to specified destination
Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

CPU Actions for Various Types of Operations



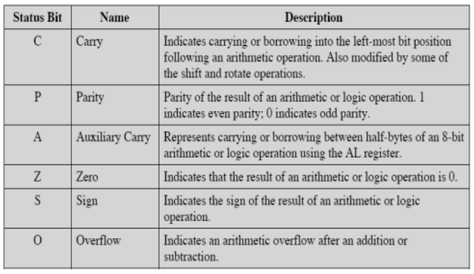
Data Transfer	Transfer data from one location to another
	If memory is involved: Determine memory address Perform virtual-to-actual-memory address transformation Check cache Initiate memory read/write
Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes and flags
Logical	Same as arithmetic
Conversion	Similar to arithmetic and logical. May involve special logic to perform conversion
Transfer of Control	Update program counter. For subroutine call/return, manage parameter passing and linkage
I/O	Issue command to I/O module
	If memory-mapped I/O, determine memory-mapped address

Pentium Operation Types



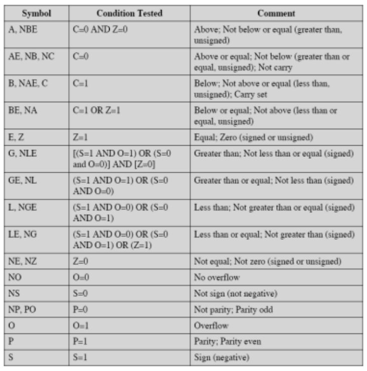
Instruction	Description
	Data Movement
MOV	Move operand, between registers or between register and memory.
PUSH	Push operand onto stack.
PUSHA	Push all registers on stack.
MOVSX	Move byte, word, dword, sign extended. Moves a byte to a word or a word to a doubleword with twos-complement sign extension.
LEA	Load effective address. Loads the offset of the source operand, rather than its value to the destination operand.
XLAT	Table lookup translation. Replaces a byte in AL with a byte from a user-coded translation table. When XLAT is executed, AL should have an unsigned index to the table. XLAT changes the contents of AL from the table index to the table entry.
IN, OUT	Input, output operand from I/O space.
	Arithmetic
ADD	Add operands.
SUB	Subtract operands.
MUL	Unsigned integer multiplication, with byte, word, or double word operands, and word, doubleword, or quadword result.
IDIV	Signed divide.
	Logical
AND	AND operands.
BTS	Bit test and set. Operates on a bit field operand. The instruction copies the current value of a bit to flag CF and sets the original bit to 1.
BSF	Bit scan forward. Scans a word or doubleword for a 1-bit and stores the number of the first 1-bit into a register.
SHL/SHR	Shift logical left or right.
SAL/SAR	Shift arithmetic left or right.
ROL/ROR	Rotate left or right.
SETcc	Sets a byte to zero or one depending on any of the 16 conditions defined by status flags.
	Control Transfer
JMP	Unconditional jump.
CALL	Transfer control to another location. Before transfer, the address of the instruction following the CALL is placed on the stack.
JE/JZ	Jump if equal/zero.
LOOPE/LOOPZ	Loops if equal/zero. This is a conditional jump using a value stored in register ECX. The instruction first decrements ECX before testing ECX for the branch condition.
INT/INTO	Interrupt/Interrupt if overflow. Transfer control to an interrupt service routine
	String Operations
MOVS	Move byte, word, dword string. The instruction operates on one element of a string, indexed by registers ESI and EDI. After each string operation, the registers are automatically incremented or decremented to point to the next element of the string.
LODS	Load byte, word, dword of string.
	High-Level Language Support
ENTER	Creates a stack frame that can be used to implement the rules of a block-structured high-level language.
LEAVE	Reverses the action of the previous ENTER.
BOUND	Check array bounds. Verifies that the value in operand 1 is within lower and upper limits. The limits are in two adjacent memory locations referenced by operand 2. An interrupt occurs if the value is out of bounds. This instruction is used to check an array index.
	Flag Control
STC	Set Carry flag.
LAHF	Load A register from flags. Copies SF, ZF, AF, PF, and CF bits into A register.
	Segment Register
LDS	Load pointer into D segment register.
	System Control
HLT	Halt.
LOCK	Asserts a hold on shared memory so that the Pentium has exclusive use of it during the instruction that immediately follows the LOCK.
ESC	Processor extension escape. An escape code that indicates the succeeding instructions are to be executed by a numeric coprocessor that supports high-precision integer and floating-point calculations.
WAIT	Wait until BUSY# negated. Suspends Pentium program execution until the processor detects that the BUSY pin is inactive, indicating that the numeric coprocessor has finished execution.
	Protection
SGDT	Store global descriptor table.
LSL	Load segment limit. Loads a user-specified register with a segment limit.
VERR/VERW	Verify segment for reading/writing.
	Cache Management
INVD	Flushes the internal cache memory.
WBINVD	Flushes the internal cache memory after writing dirty lines to memory.
INVLPG	Invalidates a translation lookaside buffer (TLB) entry.

Pentium Condition Codes



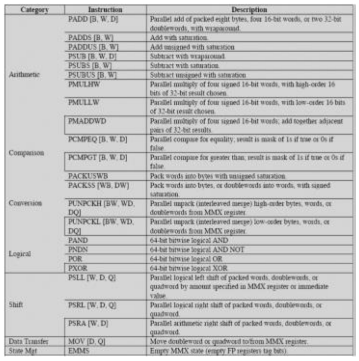
Status Bit	Name	Description
C	Carry	Indicates carrying or borrowing into the left-most bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.
P	Parity	Parity of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.
A	Auxiliary Carry	Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation using the AL register.
Z	Zero	Indicates that the result of an arithmetic or logic operation is 0.
S	Sign	Indicates the sign of the result of an arithmetic or logic operation.
O	Overflow	Indicates an arithmetic overflow after an addition or subtraction.

Pentium Conditions for Conditional Jump and SETcc Instructions



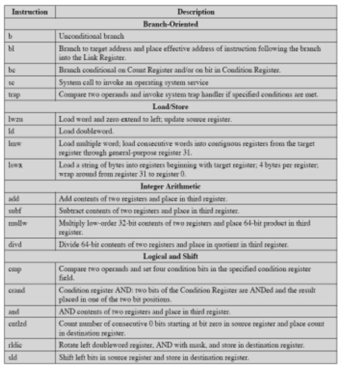
Symbol	Condition Tested	Comment
A, NBE	C=0 AND Z=0	Above; Not below or equal (greater than, unsigned)
AE, NB, NC	C=0	Above or equal; Not below (greater than or equal, unsigned); Not carry
B, NAE, C	C=1	Below; Not above or equal (less than, unsigned); Carry set
BE, NA	C=1 OR Z=1	Below or equal; Not above (less than or equal, unsigned)
E, Z	Z=1	Equal; Zero (signed or unsigned)
G, NLE	[(S=1 AND O=1) OR (S=0 and O=0)] AND [Z=0]	Greater than; Not less than or equal (signed)
GE, NL	(S=1 AND O=1) OR (S=0 AND O=0)	Greater than or equal; Not less than (signed)
L, NGE	(S=1 AND O=0) OR (S=0 AND O=1)	Less than; Not greater than or equal (signed)
LE, NG	(S=1 AND O=0) OR (S=0 AND O=1) OR (Z=1)	Less than or equal; Not greater than (signed)
NE, NZ	Z=0	Not equal; Not zero (signed or unsigned)
NO	O=0	No overflow
NS	S=0	Not sign (not negative)
NP, PO	P=0	Not parity; Parity odd
O	O=1	Overflow
P	P=1	Parity; Parity even
S	S=1	Sign (negative)

MMX Instruction Set



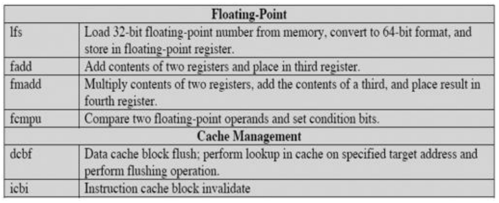
Category	Instruction	Description
Arithmetic	PADD [B, W, D]	Parallel add of packed eight bytes, four 16-bit words, or two 32-bit doublewords, with wraparound.
	PADDS [B, W]	Add with saturation.
	PADDUS [B, W]	Add unsigned with saturation
	PSUB [B, W, D]	Subtract with wraparound.
	PSUBS [B, W]	Subtract with saturation.
	PSUBUS [B, W]	Subtract unsigned with saturation
	PMULHW	Parallel multiply of four signed 16-bit words, with high-order 16 bits of 32-bit result chosen.
	PMULLW	Parallel multiply of four signed 16-bit words, with low-order 16 bits of 32-bit result chosen.
	PMADDWD	Parallel multiply of four signed 16-bit words; add together adjacent pairs of 32-bit results.
Comparison	PCMPEQ [B, W, D]	Parallel compare for equality; result is mask of 1s if true or 0s if false.
	PCMPGT [B, W, D]	Parallel compare for greater than; result is mask of 1s if true or 0s if false.
Conversion	PACKUSWB	Pack words into bytes with unsigned saturation.
	PACKSS [WB, DW]	Pack words into bytes, or doublewords into words, with signed saturation.
	PUNPCKH [BW, WD, DQ]	Parallel unpack (interleaved merge) high-order bytes, words, or doublewords from MMX register.
	PUNPCKL [BW, WD, DQ]	Parallel unpack (interleaved merge) low-order bytes, words, or doublewords from MMX register.
Logical	PAND	64-bit bitwise logical AND
	PNDN	64-bit bitwise logical AND NOT
	POR	64-bit bitwise logical OR
	PXOR	64-bit bitwise logical XOR
Shift	PSLL [W, D, Q]	Parallel logical left shift of packed words, doublewords, or quadword by amount specified in MMX register or immediate value.
	PSRL [W, D, Q]	Parallel logical right shift of packed words, doublewords, or quadword.
	PSRA [W, D]	Parallel arithmetic right shift of packed words, doublewords, or quadword.
Data Transfer	MOV [D, Q]	Move doubleword or quadword to/from MMX register.
State Mgt	EMMS	Empty MMX state (empty FP registers tag bits).

PowerPC Operation Types



Instruction	Description
Branch-Oriented	
b	Unconditional branch
bl	Branch to target address and place effective address of instruction following the branch into the Link Register.
bc	Branch conditional on Count Register and/or on bit in Condition Register.
sc	System call to invoke an operating system service
trap	Compare two operands and invoke system trap handler if specified conditions are met.
Load/Store	
lwzu	Load word and zero extend to left; update source register.
ld	Load doubleword.
lmw	Load multiple word; load consecutive words into contiguous registers from the target register through general-purpose register 31.
lswx	Load a string of bytes into registers beginning with target register; 4 bytes per register; wrap around from register 31 to register 0.
Integer Arithmetic	
add	Add contents of two registers and place in third register.
subf	Subtract contents of two registers and place in third register.
mullw	Multiply low-order 32-bit contents of two registers and place 64-bit product in third register.
divd	Divide 64-bit contents of two registers and place in quotient in third register.
Logical and Shift	
cmp	Compare two operands and set four condition bits in the specified condition register field.
crand	Condition register AND: two bits of the Condition Register are ANDed and the result placed in one of the two bit positions.
and	AND contents of two registers and place in third register.
cntlzd	Count number of consecutive 0 bits starting at bit zero in source register and place count in destination register.
rldic	Rotate left doubleword register, AND with mask, and store in destination register.
sld	Shift left bits in source register and store in destination register.

PowerPC Operation Types



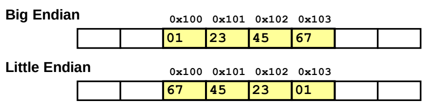
Floating-Point	
lfs	Load 32-bit floating-point number from memory, convert to 64-bit format, and store in floating-point register.
fadd	Add contents of two registers and place in third register.
fmadd	Multiply contents of two registers, add the contents of a third, and place result in fourth register.
fcmpu	Compare two floating-point operands and set condition bits.
Cache Management	
dcbf	Data cache block flush; perform lookup in cache on specified target address and perform flushing operation.
icbi	Instruction cache block invalidate

Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
- Some conventions
  - Sun's, Mac's are "Big Endian" machines
    - Least significant byte has highest address
  - Alphas, PC's are "Little Endian" machines
    - Least significant byte has lowest address

Byte Ordering Example

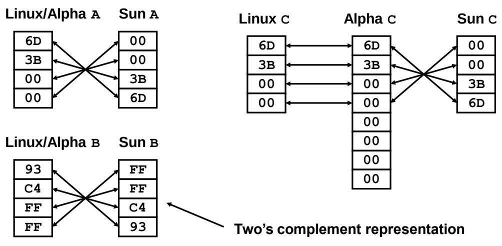
- Big Endian
  - Least significant byte has highest address
- Little Endian
  - Least significant byte has lowest address
- Example
  - Variable `x` has 4-byte representation `0x01234567`
  - Address given by `&x` is `0x100`



Representing Integers

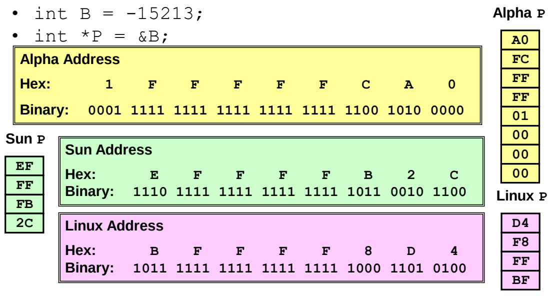
- `int A = 15213;`
- `int B = -15213;`
- `long int C = 15213;`

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D



Representing Pointers

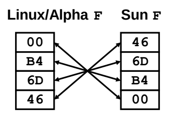
- `int B = -15213;`
- `int *P = &B;`



Different compilers & machines assign different locations to objects

Representing Floats

- `Float F = 15213.0;`



IEEE Single Precision Floating Point Representation

Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000
15213:			1110	1101	1011	01		

Not same as integer representation, but consistent across machines

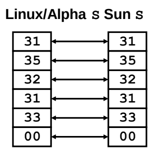
Can see some relation to integer representation, but not obvious

49

Representing Strings

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code `0x30`
      - Digit *i* has code `0x30`+*i*
  - String should be null-terminated
    - Final character = 0
- Compatibility
  - Byte ordering is not an issue
    - Data are single byte quantities
  - Text files generally platform independent
    - Except for different conventions of line termination character(s)!

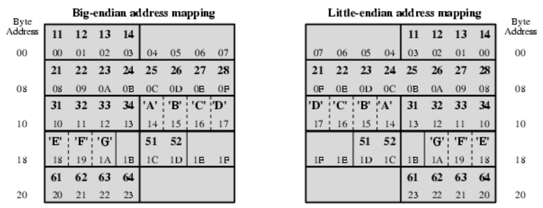
- `char S[6] = "15213";`



50

Example of C Data Structure

```
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

51

Common file formats and their endian order

- **Adobe Photoshop** -- Big Endian
- **BMP (Windows and OS/2 Bitmaps)** -- Little Endian
- **DXF (AutoCad)** -- Variable
- **GIF** -- Little Endian
- **IMG (GEM Raster)** -- Big Endian
- **JPEG** -- Big Endian
- **FLI (Autodesk Animator)** -- Little Endian
- **MacPaint** -- Big Endian
- **PCX (PC Paintbrush)** -- Little Endian
- **PostScript** -- Not Applicable (text!)
- **POV (Persistence of Vision ray-tracer)** -- Not Applicable (text!)
- **QTM (Quicktime Movies)** -- Little Endian (on a Mac!)
- **Microsoft RIFF (.WAV & .AVI)** -- Both
- **Microsoft RTF (Rich Text Format)** -- Little Endian
- **SGI (Silicon Graphics)** -- Big Endian
- **Sun Raster** -- Big Endian
- **TGA (Targa)** -- Little Endian
- **TIFF** -- Both, Endian identifier encoded into file
- **WPG (WordPerfect Graphics Metafile)** -- Big Endian (on a PC!)
- **XWD (X Window Dump)** -- Both, Endian identifier encoded into file

52

Addressing Modes and Formats

53

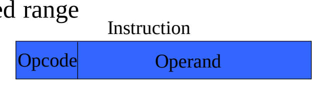
Addressing Modes

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

54

Immediate Addressing

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



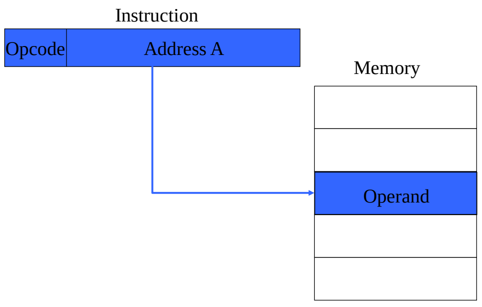
55

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

56

Direct Addressing Diagram



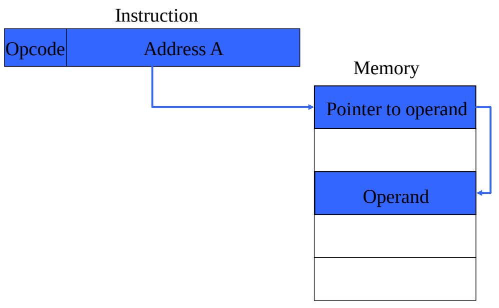
57

Indirect Addressing (1)

- Memory cell pointed to by address field contains the address of (pointer to) the operand
- Large address space
- 2^n where n = word length
- May be nested, multilevel, cascaded
- Multiple memory accesses to find operand
- Hence slower

58

Indirect Addressing Diagram



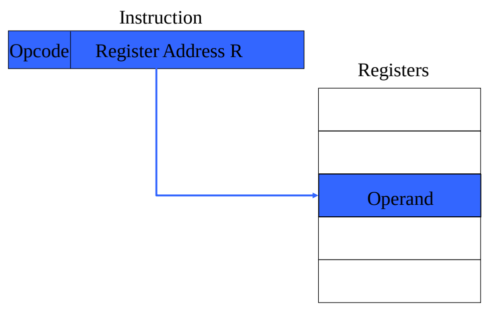
59

Register Addressing (1)

- Operand is held in register named in address field
- 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

60

Register Addressing Diagram



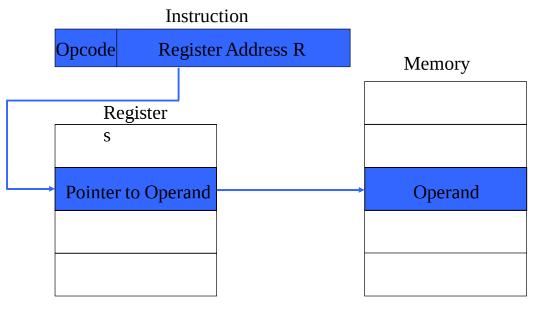
61

Register Indirect Addressing

- Operand is held in memory cell pointed to by contents of register R named in address field
- EA = (R)
- Large address space (2^n)
- One fewer memory access than indirect addressing

62

Register Indirect Addressing Diagram



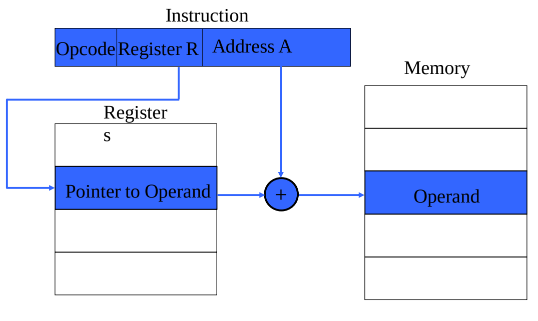
63

Displacement Addressing

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

64

Displacement Addressing Diagram



65

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

66

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++

Combinations

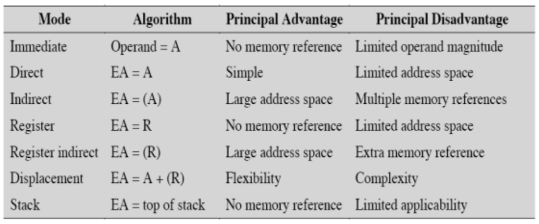
- Postindex
- EA = (A) + (R)

- Preindex
- EA = (A+(R))

Stack Addressing

- Operand is (implicitly) on top of stack
- e.g.
  - ADD   Pop top two items from stack and add

Summary of basic addressing modes



Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

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

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

73

Allocation of Bits

- Number of addressing modes
- Number of operands
- Register versus memory
- Number of register sets
- Address range
- Address granularity

74

75