

25-Sep-06 (1)



CSC2510 - Computer Organization

Lecture 3: Memory

Philip Leong

25-Sep-06 (2)



Memory

- **Announcement - TUTS: Wed 1:30pm LHCG06, Thur 4:30pm LHC101**
- This lecture: focus on memory and addressing memory
 - In particular RAM (which is primary storage)
- Consists of storage cells
 - Arranged as **bits** (0 or 1)
 - We can deal with them in n-bit groups called **words** (typically 8, 16, 32 or 64 bits)
 - Sometimes use **bytes** which are 8-bits in length e.g. for characters
 - $1024=1K$
 - $1024*1024=1M$
 - $1024*1024*1024=1G$
- Usually refer to memory size in bytes e.g. we say we have 128MB memory and rarely use words as the unit

25-Sep-06 (3)



Addresses

- Use **addresses** to store or retrieve a single item of information
- For some k, memory consists of 2^k unique addresses which range from $0-2^k-1$
 - The possible addresses are the **address space** of the computer
 - E.g. 28-bit address has 2^{28} (268435456) locations (normally we use words for addresses c.f. memory size on previous slide)

25-Sep-06 (4)



Quiz

- Given the information about the machine below from Dell's website, how many unique 32-bit locations does it have? How many bits? What does the 2x512 mean?

Online Price HKD 5,899.00

Base System Dimension(TM) 9200 Intel(R) Pentium(R) D915 Processor with Dual Core Technology

Memory 1GB (2x512) NECC Dual Channel DDR2 667MHz SDRAM Memory

25-Sep-06 (5)



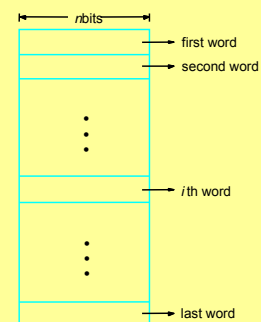
Byte addresses

- Information quantities: bit, byte, word
- Byte=8 bits
- Word typically varies 16-64 bits (the IA-32 architecture has 32-bit words which we will assume from now on)
- Most machines address memory in units of bytes
 - Implies for a 32-bit machine, successive words are at address 0, 4, 8, 12 ...

25-Sep-06 (6)



Organization of memory



25-Sep-06 (7)

Integers and Characters

The diagram shows a 32-bit word. The leftmost bit is labeled b_{31} and is identified as the sign bit. The remaining bits are labeled b_{30} , ..., b_1 , b_0 . Below this, a 32-bit word is divided into four 8-bit segments, each labeled "ASCII character".

Sign bit: $b_{31} = 0$ for positive numbers
 $b_{31} = 1$ for negative numbers

(a) A signed integer

(b) Four characters

25-Sep-06 (8)

More/less significant bytes

- Consider the hexadecimal (base 16) 32-bit number $12342A3F = 1 \times 16^7 + 2 \times 16^6 + 3 \times 16^5 + 4 \times 16^4 + 2 \times 16^3 + 10 \times 16^2 + 3 \times 16^1 + 15 \times 16^0$
- This number has four bytes 12, 34, 2A, 4F (4x8=32-bits)
- Bytes/bits with higher weighting are "more significant" e.g. the byte 34 is more significant than 2A
- Bytes/bits with lower weighting are "less significant"
- We also use terms "most significant byte/bit" and "least significant byte/bit"

25-Sep-06 (9)

Big/little endian

- Two ways byte addresses can be assigned across words
 - more significant bytes first (big endian)
 - less significant bytes first (little endian)

25-Sep-06 (10)

Big/little endian

The diagram compares two memory layouts. On the left, labeled "(a) Big-endian assignment", a word at address $2^k - 4$ contains bytes at addresses $2^k - 4$, $2^k - 3$, $2^k - 2$, and $2^k - 1$. On the right, labeled "(b) Little-endian assignment", the same word contains bytes at addresses $2^k - 1$, $2^k - 2$, $2^k - 3$, and $2^k - 4$.

(a) Big-endian assignment

(b) Little-endian assignment

25-Sep-06 (11)

Word alignment

- 32-bit words align naturally at addresses 0, 4, 8 ...
 - These are aligned addresses
- Unaligned accesses are either not allowed or slower e.g. read a 32-bit word from address 1 (why?)
- What about for 16 and 64 bit words?

25-Sep-06 (12)

Representing strings

- All data so far (bits, bytes, words) are of known length
- How can we represent strings which could be variable length?
 - E.g. "University"?

25-Sep-06 (13)



Representing strings

- Method 1:
 - Use a null to mark the end of the string
 - `'U','n','i','v','e','r','s','i','t','y',0`
- Method 2:
 - Use a separate number to represent the length
 - `10,'U','n','i','v','e','r','s','i','t','y'`

25-Sep-06 (14)



Memory Operations

- Computer instructions serve to transfer and manipulate data from memory
- Two operations
 - Load (read or fetch): processor sends address to memory, memory returns data e.g. $R1 \leftarrow [LOC]$ (R1 is an internal register in the processor)
 - Store (write): processor sends address and data, memory overwrites location with new data e.g. $[LOC] \leftarrow R1$

25-Sep-06 (15)



Instructions

- Computer operates by executing a sequence of instructions
- Could perform one of the following
 - Data transfer between processor and memory
 - Arithmetic and logical operations on data in processor
 - Program sequencing and control (i.e. branches, subroutine calls)
 - I/O transfers

25-Sep-06 (16)



Assembly language

- Three address instructions:
 - “op src1, src2, dst” e.g. “add a,b,c” means “ $c \leftarrow [a] + [b]$ ”
- Two address instructions:
 - Some computers e.g. IA-32 only use 2 operands e.g. “add a,b” means “ $b \leftarrow [a] + [b]$ ”
 - Operand b is both a source and destination
- One address instructions (next slide)
- **Note: some machines (e.g. IA-32) put destination registers first e.g. “op dst, src”**
- What does “move a,b” do?
- How do I compute “add a,b,c” on a two address machine?

25-Sep-06 (17)



One address instructions

- In some machines, there is an implicit register called an accumulator (A)
 - “add b” means “ $A \leftarrow A + [b]$ ”
 - “load b” means “ $A \leftarrow [b]$ ”
 - “store b” means “ $[b] \leftarrow A$ ”
- What are some of the advantages and disadvantages of one/two/three address instructions?
 - Think in terms of power as well as number of bits needed to specify all the operands

25-Sep-06 (18)



Registers

- Most higher end machines have a number of **registers** to store temporary data in the processor
 - Transfers to/from memory (i.e. Load/Store) are relatively slow
 - Operations only involving registers are fast
 - High speed makes time to execute an instruction shorter
 - In many computers (e.g. IA-32), only one instruction can be from memory and the others need to be in registers
 - e.g. “Add a,b” and “Move a,b” are not allowed on IA-32
- | | |
|-------|------|
| Load | A,R1 |
| Add | B,R1 |
| Store | R1,C |
- How would we do “Move a,b”?

25-Sep-06 (19)



Instruction execution

- Assume 32-bit words
- Memory is byte addressable
- Wish to compute “ $c \leftarrow [a] + [b]$ ”
 - Move A,R0
 - Add B,R0
 - Move R0,C

25-Sep-06 (20)



A program in memory

- Memory holds words that represent instructions
- Program counter (PC) holds address of instruction to be executed next
- Processor control circuits fetch and execute instruction at the PC
 - Instruction fetch $IR \leftarrow [PC]$, $PC = PC + 1$
 - Instruction execute – look at IR and take the appropriate action

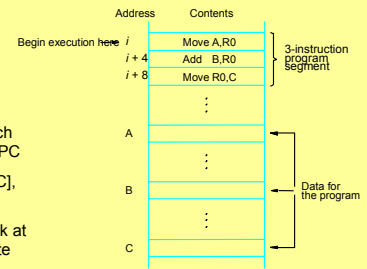
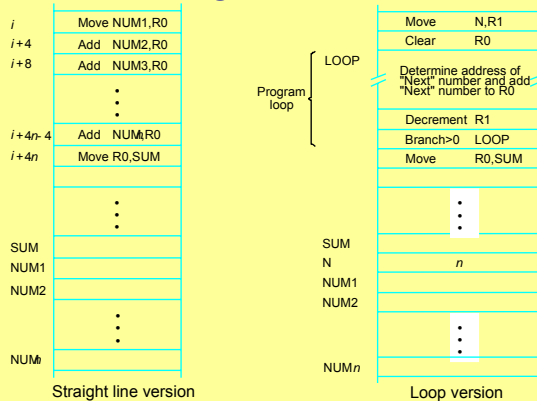


Figure 2.8. A program for $C \leftarrow [A] + [B]$.

25-Sep-06 (21)



Adding N numbers



25-Sep-06 (22)



Condition codes

- In order to do **conditional branches** and other instructions, operations implicitly set **flags**
- Four commonly used (1-bit) flags
 - N (negative) 1 if result –ve else 0
 - Z (zero) 1 if result 0 else 0
 - V (overflow) 1 if arithmetic overflow occurs else 0
 - C (carry) 1 if carry out occurs –ve else 0
- E.g. Add R1,R2 could generate any of these flags
 - Give values for R1 and R2 (assuming they are 32-bit registers) for each case

25-Sep-06 (23)



Examples

- N (negative)
 - R1=2, R2=-5
- Z (zero)
 - R1=10, R2=-10
- V (overflow)
 - R1=7FFFFFFF, R2=1
- C (carry)
 - R1=FFFFFFFF, R2=1

25-Sep-06 (24)



Addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R i	EA = R i
Absolute (Direct)	LOC	EA = LOC
Indirect	(R i) (LOC)	EA = [R i] EA = [LOC]
Index	X(R i)	EA = [R i] + X
Base with index	(R i,R j)	EA = [R i] + [R j]
Base with index and offset	X(R i,R j)	EA = [R i] + [R j] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R i)+	EA = [R i]; Increment R i
Autodecrement	-(R i)	Decrement R i; EA = [R i]

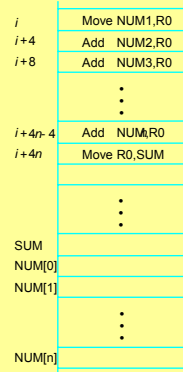
EA = effective address
Value = a signed number

25-Sep-06 (25)



Variables and Constants

- Register mode – operand is in a register
e.g. "add R1,R2" ($R2 \leftarrow R1+R2$)
- Absolute/direct mode – operand is in a memory location, address is in instruction
e.g. "Move SUM, R0" ($R0 \leftarrow [SUM]$)
 - If you declare a global variable in a high level language e.g. int a,b; compiler will allocate space in memory for this variable and access via absolute mode
 - NOTE: SUM is a symbolic name representing a memory address
- Immediate mode – Move #200, R0 ($R0 \leftarrow 200$)
 - How do I write $A=B+6$?

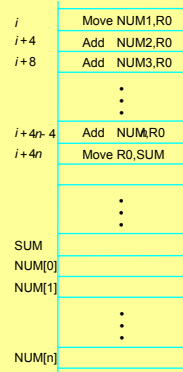


25-Sep-06 (26)



Indirect addressing

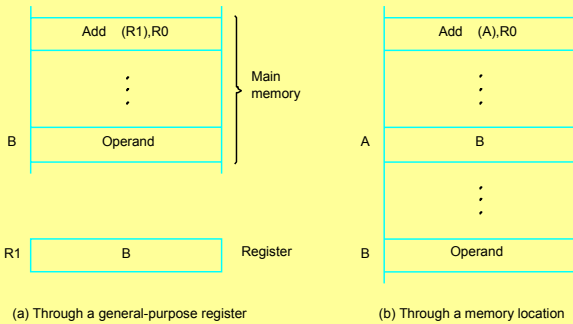
- Indirect mode – EA is contents of register (or memory location) whose address is in instruction
e.g. "Add (R0),R1" ($R1 \leftarrow R1+[R0]$)
- Array access NUM[i]
 - Suppose value of i is in register R1
 - R2 points to NUM[0]
 - How do I read NUM[i]?



25-Sep-06 (27)



Indirect addressing



25-Sep-06 (28)



Indirect addressing example

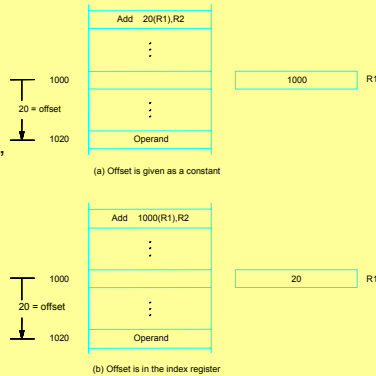
Address	Contents	
	Move N,R1	} Initialization
	Move #NUM1,R2	
	Clear R0	
	Add (R2),R0	
LOOP	Add #4,R2	
	Decrement R1	
	Branch>0 LOOP	
	Move R0,SUM	

25-Sep-06 (29)



Indexed addressing

- Index mode – indirect mode plus an offset
e.g. "Add X(R0),R1" ($R1 \leftarrow R1+[R0+X]$)
– i.e. $EA=X+[Ri]$



25-Sep-06 (30)



What does this do?

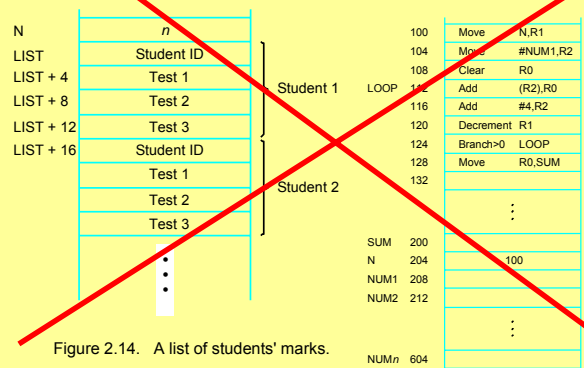


Figure 2.14. A list of students' marks.