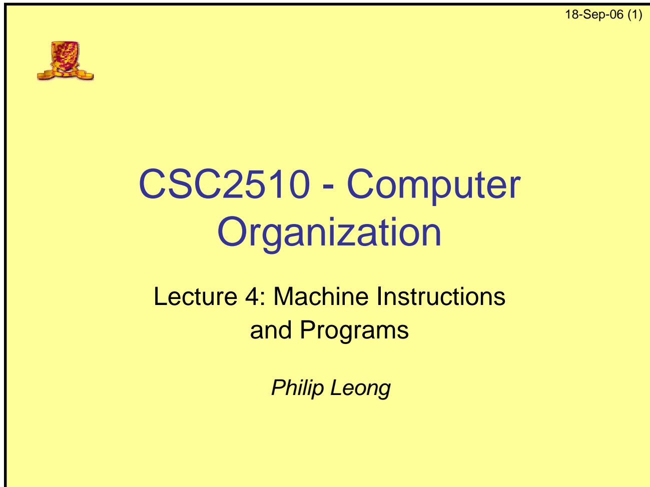
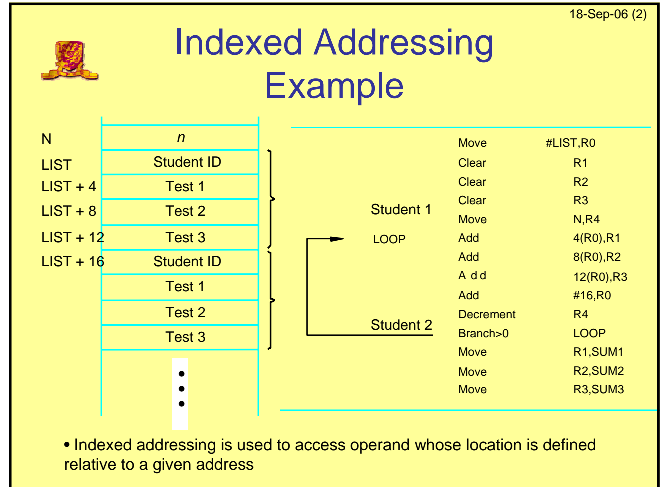
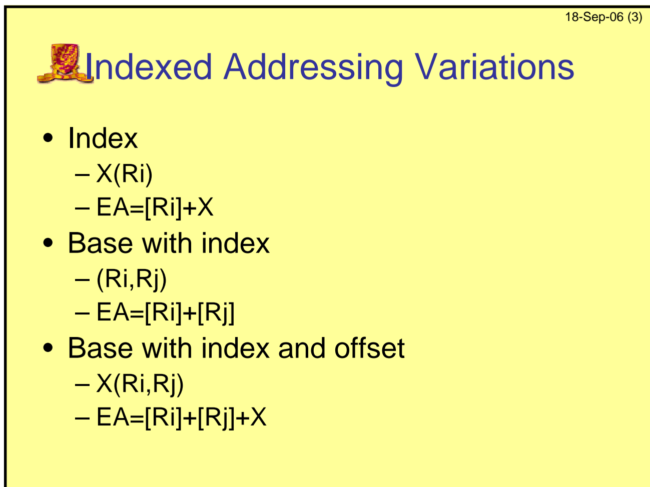
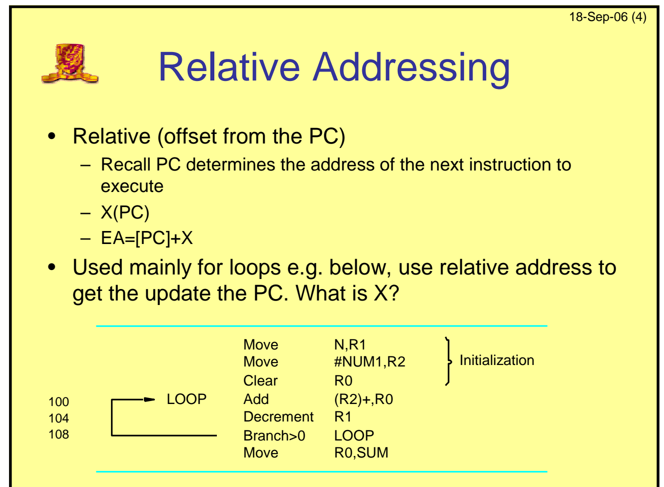
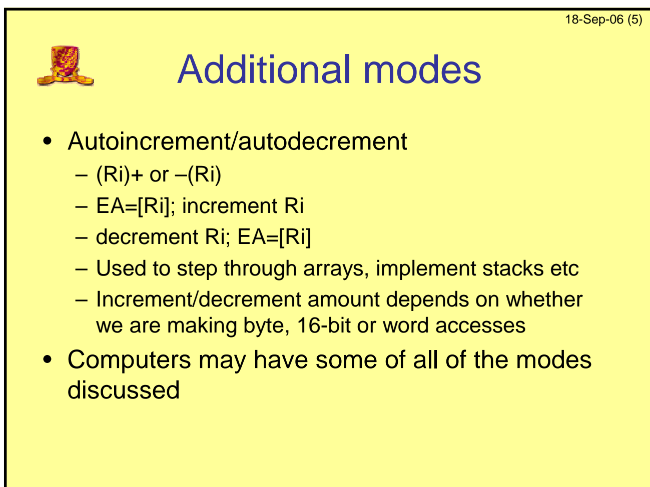
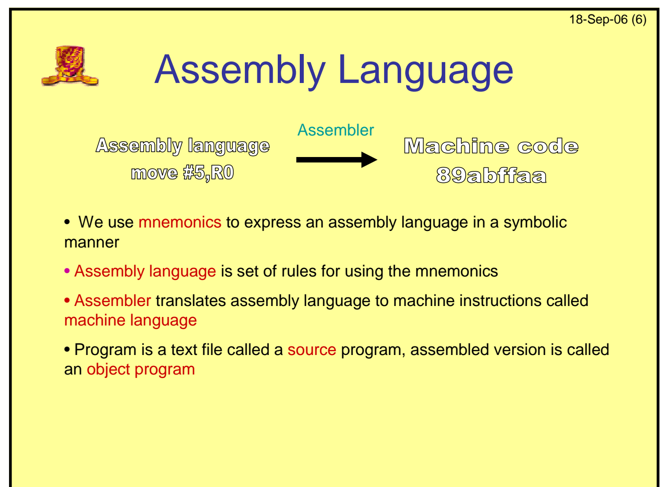

18-Sep-06 (1)
CSC2510 - Computer Organization
Lecture 4: Machine Instructions and Programs
Philip Leong


18-Sep-06 (2)
Indexed Addressing Example
N
LIST
LIST + 4
LIST + 8
LIST + 12
LIST + 16
n
Student ID
Test 1
Test 2
Test 3
Student ID
Test 1
Test 2
Test 3
Student 1
Student 2
LOOP
Move #LIST,R0
Clear R1
Clear R2
Clear R3
Move N,R4
Add 4(R0),R1
Add 8(R0),R2
Add 12(R0),R3
Add #16,R0
Decrement R4
Branch>0 LOOP
Move R1,SUM1
Move R2,SUM2
Move R3,SUM3
• Indexed addressing is used to access operand whose location is defined relative to a given address


18-Sep-06 (3)
Indexed Addressing Variations
• Index
– X(Ri)
– EA=[Ri]+X
• Base with index
– (Ri,Rj)
– EA=[Ri]+[Rj]
• Base with index and offset
– X(Ri,Rj)
– EA=[Ri]+[Rj]+X


18-Sep-06 (4)
Relative Addressing
• Relative (offset from the PC)
– Recall PC determines the address of the next instruction to execute
– X(PC)
– EA=[PC]+X
• Used mainly for loops e.g. below, use relative address to get the update the PC. What is X?
Move N,R1
Move #NUM1,R2
Clear R0
Initialization
100 LOOP Add (R2)+,R0
104 Decrement R1
108 Branch>0 LOOP
Move R0,SUM


18-Sep-06 (5)
Additional modes
• Autoincrement/autodecrement
– (Ri)+ or –(Ri)
– EA=[Ri]; increment Ri
– decrement Ri; EA=[Ri]
– Used to step through arrays, implement stacks etc
– Increment/decrement amount depends on whether we are making byte, 16-bit or word accesses
• Computers may have some of all of the modes discussed


18-Sep-06 (6)
Assembly Language
Assembly language
move #5,R0
Assembler
Machine code
89abffaa
• We use mnemonics to express an assembly language in a symbolic manner
• Assembly language is set of rules for using the mnemonics
• Assembler translates assembly language to machine instructions called machine language
• Program is a text file called a source program, assembled version is called an object program

Opcodes

- Mnemonic Move R0,SUM

	INSTRUCTION	AMODE1	AMODE2	OPERAND1	OPERAND2
	MOV	REG	ABS	0	SUM
	F	1	2	0	FF89
#bits	4	4	4	4	16

Assembly language

- Must have syntax to explain what mode is being used
  - E.g. ADD 5,R5
  - does the 5 mean immediate or absolute?
  - ADD #5,R5 or ADDI 5,R5

    Immediate
  - Indirect addressing normally specified by parentheses e.g. move #5,(R2)

Assembler directives

- STUDENTS EQU 20
  - STUDENTS is a symbol meaning 20 i.e. a label
  - No machine code is generated for this directive
- ORIGIN 200
  - Specifies that the assembler should place machine code at address 200
- DATAWORD 100
  - Data value 100 should be placed in memory
- Labels allow symbolic references to memory addresses
  - Don't need to know their actual value

Assembly language vs machine code

Label	Address	Instruction	Operands
	100	Move	N,R1
	104	Move	#NUM1,R2
	108	Clear	R0
LOOP	112	Add	(R2),R0
	116	Add	#4,R2
	120	Decrement	R1
	124	Branch>0	LOOP
	128	Move	R0,SUM
	132		
		⋮	
SUM	200		
N	204	100	
NUM1	208		
NUM2	212		
		⋮	
NUMn	604		

	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

Assembler

- Has to know
  - How to interpret machine language (directives, instructions, addressing modes etc)
  - Where to place the instructions in memory
  - Where to place the data in memory
- Scans through source program, keeps track of all names and corresponding numerical values in symbol table e.g. what all the labels mean
- Calculate branch addresses
  - Forward branch problem – how can it work out forward addresses?

Two pass assembler

- First pass
  - Work out all the addresses of labels
- Second pass
  - Generate machine code, substituting values for the labels

Loader

- Transfers machine code from disk to memory
- Execute first instruction

Number notation

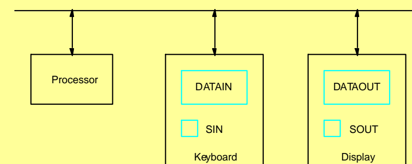
- Differs with different assemblers
- Need to be able to specify constants in binary, decimal, hex
  - ADD #93, R1
  - ADD #%01011101, R1
  - ADD #$5D, R1

Basic I/O

- I/O is the means by which data are transferred between the processor and the outside world
- Devices operate at different speeds to the processor so handshaking is required

Keyboard/display Example

- The keyboard and display are coordinated via software
- Register (on device) assigned to the keyboard hardware
  - DATAIN contains ASCII of last typed character
  - SIN is the status control flag, normally 0. When a character typed, becomes 1. After the processor reads DATAIN, it is automatically set back to 0
- Register (on device) assigned to the display hardware
  - DATAOUT receives a character code
  - SOUT is the status control flag. It is 1 when ready to receive a character, set to 0 when the character is being transferred
- These registers form the respective device interface



Programmed IO

```
READWAIT   Branch to READWAIT if SIN=0
           INPUT from DATAIN to R1

WRITEWAIT  Branch to WRITEWAIT if SOUT=0
           Output from R1 to DATAOUT
```

Memory Mapped IO

- On many machines, registers such as DATAIN, DATAOUT are memory-mapped
  - Read and write specific memory locations to communicate with device
  - MoveByte DATAIN,R1
  - MoveByte R1,DATAOUT
- SIN and SOUT might be bits in a device status register e.g. bit 3

Memory-Mapped IO

READWAIT Branch to READWAIT if SIN=0
INPUT from DATAIN to R1

```
READWAIT  Testbit    #3,INSTATUS
          Branch=0   READWAIT
          MoveByte   DATAIN,R1
```

What about WRITEWAIT?

WRITEWAIT Branch to WRITEWAIT if SOUT=0
Output from R1 to DATAOUT

Complete Example

	Move	#LOC,R0	Initialize pointer register R0 to point to the address of the first location in memory where the characters are to be stored.
READ	TestBit	#3,INSTATUS	Wait for a character to be entered
	Branch=0	READ	in the keyboard buffer DATAIN.
	MoveByte	DATAIN,(R0)	Transfer the character from DATAIN into the memory (this clears SIN to 0).
ECHO	TestBit	#3,OUTSTATUS	Wait for the display to become ready.
	Branch=0	ECHO	
	MoveByte	(R0),DATAOUT	Move the character just read to the display buffer register (this clears SOUT to 0).
	Compare	#CR,(R0)+	Check if the character just read is CR (carriage return). If it is not CR, then
	Branch≠0	READ	branch back and read another character. Also, increment the pointer to store the next character.

Stacks

- List of data elements (usually bytes or words)
  - Elements can only be removed at one end of the list
  - Last-in-first-out
- Can be implemented in several ways, one way is
  - First element placed in BOTTOM
  - Grows in direction of decreasing memory address
  - Assume 32-bit data

Stack Implementation

```
Subtract #4,SP
Move     NEWITEM,(SP)   ; push

Move     (SP),ITEM      ; pop
Add #4,SP
```

With autoincrement and autodecrement

```
Move     NEWITEM,-(SP)  ; push
```

1. How do you write pop using autoincrement?
2. How can I check that push/pop doesn't overflow/underflow?

(a) After push from NEWITEM

(b) After pop into ITEM

Safe pop/push

SAFEPOP	Compare	#2000,SP	Check to see if the stack pointer contains an address value greater than 2000. If it
	Branch>0	EMPTYERROR	does, the stack is empty. Branch to the routine EMPTYERROR for appropriate action.
	Move	(SP)+,ITEM	Otherwise, pop the top of the stack into memory location ITEM.

(a) Routine for a safe pop operation

SAFEPUSH	Compare	#1500,SP	Check to see if the stack pointer contains an address value equal
	Branch≤0	FULLERROR	to or less than 1500. If it does, the stack is full. Branch to the routine FULLERROR for appropriate action.
	Move	NEWITEM, –(SP)	Otherwise, push the element in memory location NEWITEM onto the stack.

Similar data structures

- Queue
  - First-in-first-out
  - Unlike a stack, need to keep track of both the front and end for removal and insertion respectively
  - Need two pointers to keep track of both ends
  - Assuming it moves through memory in direction of higher addresses, as it is used, it walks through memory towards higher addresses
- Circular buffers avoid this problem by limiting to a fixed region in memory
  - Start at BEGINNING and entries appended until it reaches END after which it wraps back around to BEGINNING
  - Need to deal with cases when it is completely full and completely empty

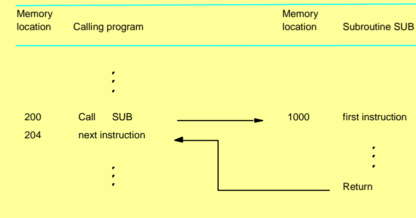
18-Sep-06 (25)

Subroutines

- Often need to perform subtask on different data. Subtask called a subroutine
- Rather than include the same sequence of instructions everywhere it is needed, call a subroutine instead
  - One copy of subroutine stored in memory
  - Subroutine call causes a branch to the subroutine
  - At the end of the subroutine, a return instruction is executed
  - Program resumes execution at the instruction immediately following the subroutine call

18-Sep-06 (26)

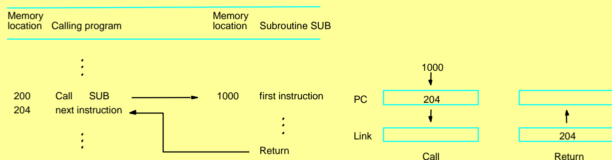
Subroutine call



18-Sep-06 (27)

Implementation

- Since subroutine can be called from a number of different places in the program, need to keep track of the return address
  - Call instruction saves the contents of the PC
  - Simplest is a link register



18-Sep-06 (28)

Call Sequence

- Call
  - Store the contents of the PC in link register
  - Branch to target address specified in the instruction
- Return
  - Branch to address contained in the link register

What about the case of nested subroutines (i.e. a subroutine calls a subroutine)?

What data structure do we need?

18-Sep-06 (29)

Nested Subroutines

- Call
  - Push the contents of the PC to the processor stack (pointed to by the stack point SP)
  - Branch to target address specified in the instruction
- Return
  - Branch to address popped from processor stack