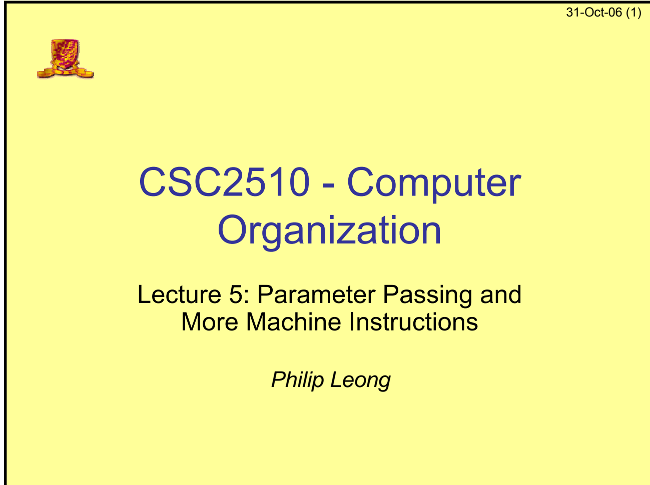
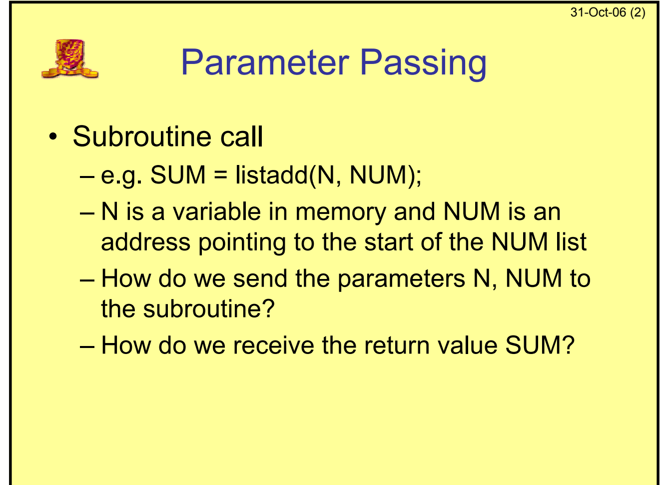
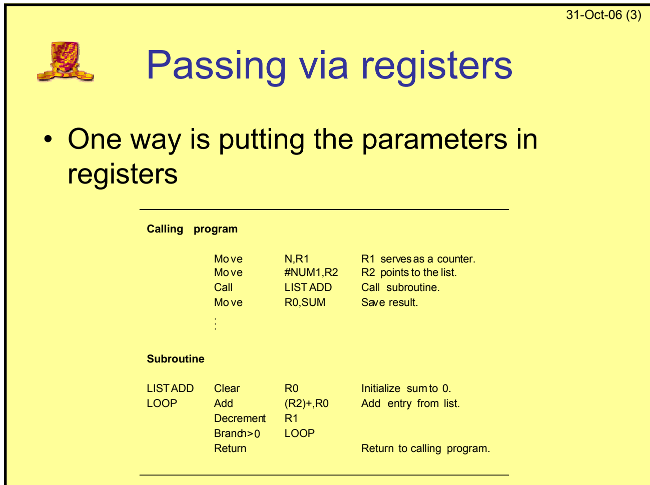
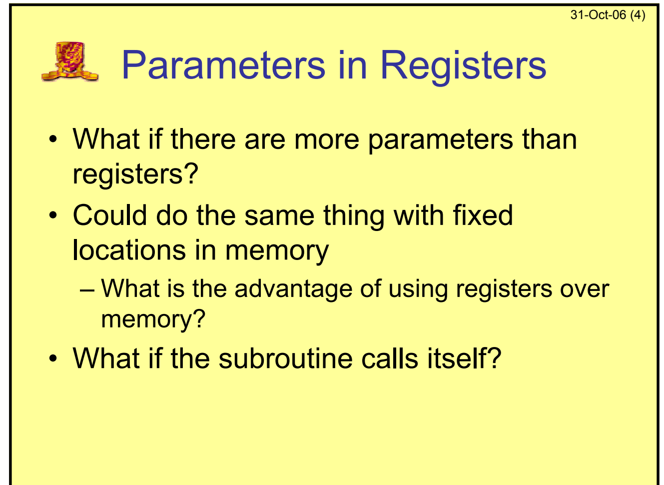
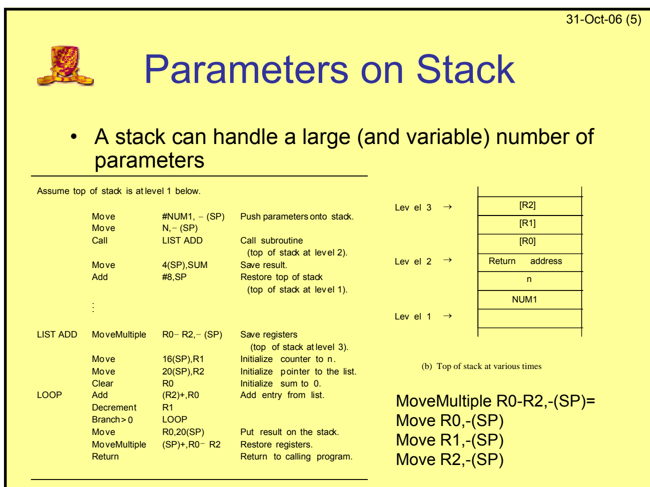
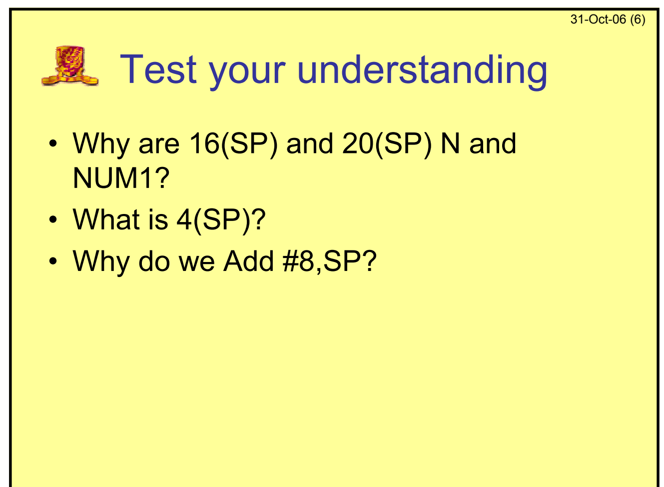

31-Oct-06 (1)
CSC2510 - Computer Organization
Lecture 5: Parameter Passing and More Machine Instructions
Philip Leong


31-Oct-06 (2)
Parameter Passing
• Subroutine call
– e.g. SUM = listadd(N, NUM);
– N is a variable in memory and NUM is an address pointing to the start of the NUM list
– How do we send the parameters N, NUM to the subroutine?
– How do we receive the return value SUM?


31-Oct-06 (3)
Passing via registers
• One way is putting the parameters in registers
Calling program
Move N,R1 R1 serves as a counter.
Move #NUM1,R2 R2 points to the list.
Call LISTADD Call subroutine.
Move R0,SUM Save result.
⋮
Subroutine
LISTADD Clear R0 Initialize sum to 0.
LOOP Add (R2)+,R0 Add entry from list.
Decrement R1
Branch>0 LOOP
Return Return to calling program.


31-Oct-06 (4)
Parameters in Registers
• What if there are more parameters than registers?
• Could do the same thing with fixed locations in memory
– What is the advantage of using registers over memory?
• What if the subroutine calls itself?


31-Oct-06 (5)
Parameters on Stack
• A stack can handle a large (and variable) number of parameters
Assume top of stack is at level 1 below.
Move #NUM1, −(SP) Push parameters onto stack.
Move N,−(SP)
Call LIST ADD Call subroutine (top of stack at level 2).
Move 4(SP),SUM Save result.
Add #8,SP Restore top of stack (top of stack at level 1).
⋮
LIST ADD MoveMultiple R0−R2,−(SP) Save registers (top of stack at level 3).
Move 16(SP),R1 Initialize counter to n.
Move 20(SP),R2 Initialize pointer to the list.
Clear R0 Initialize sum to 0.
LOOP Add (R2)+,R0 Add entry from list.
Decrement R1
Branch>0 LOOP
Move R0,20(SP) Put result on the stack.
MoveMultiple (SP)+,R0−R2 Restore registers.
Return Return to calling program.
Level 3 →
[R2]
[R1]
[R0]
Level 2 →
Return address
n
NUM1
Level 1 →
(b) Top of stack at various times
MoveMultiple R0-R2,-(SP)=
Move R0,-(SP)
Move R1,-(SP)
Move R2,-(SP)


31-Oct-06 (6)
Test your understanding
• Why are 16(SP) and 20(SP) N and NUM1?
• What is 4(SP)?
• Why do we Add #8,SP?

Passing Parameters on Stack

- In previous example, stack used to
  - Pass parameters
  - Store subroutine return address
  - Save and restore registers R0-R2 which are used in the subroutine (R0=SUM, R1=N, R2=NUM1)
- Need to be careful to push/pop everything in the right order

Passing by value and reference

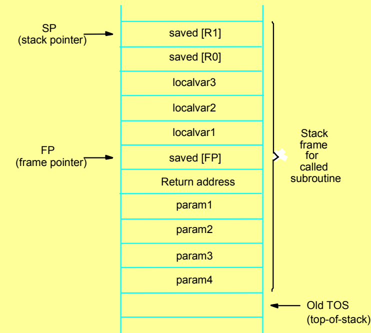
- Passing by reference
  - Instead of passing the actual values in the list, the routine passes the address of the NUM list
- The actual number N is passed by value

Stack frame

- In previous example, 6 stack entries are used in the subroutine
- In a stack frame we create a private work space for the subroutine
  - Created upon entry
  - Freed up upon exit
  - Calling parameters, return address, saved registers as before
  - Local memory variables can also be allocated on the stack

Stack frame

- Frame pointer (FP) provides convenient access to parameters and local variables
- First 2 instructions in subroutine
  - Move FP,-(SP)
  - Move SP,FP
- Where does SP point after this?
- What would Subtract #12,SP do?
- How do I reference the 2nd local variable?
- How do I restore the R0,R1 registers before exit?



Example

Main program

```
                 :
2000          Move           PARAM2, – (SP)   Place parameters on stack.
2004          Move           PARAM1, – (SP)
2008          Call           SUB1
2012          Move           (SP),RESULT      Store result.
2016          Add            #8,SP            Restore stack level.
2020          next instruction
                 :
```

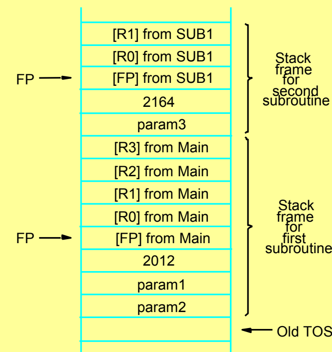
First subroutine

```
2100   SUB1   Move           FP,– (SP)        Save frame pointer register.
2104          Move           SP,FP            Load the frame pointer.
2108          MoveMultiple   R0– R3,– (SP)    Save registers.
2112          Move           8(FP),R0         Get first parameter.
              Move           12(FP),R1        Get second parameter.
                 :
              Move           PARAM3, – (SP)   Place a parameter on stack.
2160          Call           SUB2
2164          Move           (SP)+,R2         Pop SUB2 result into R2.
                 :
              Move           R3,8(FP)         Place answer on stack.
              MoveMultiple   (SP)+,R0– R3     Restore registers.
              Move           (SP)+,FP         Restore frame pointer register.
              Return                          Return to Main program.
```

Second subroutine

```
3000   SUB2   Move           FP,– (SP)        Save frame pointer register.
              Move           SP,FP            Load the frame pointer.
              MoveMultiple   R0– R1,– (SP)    Save registers R0 and R1.
              Move           8(FP),R0         Get the parameter.
                 :
              Move           R1,8(FP)         Place SUB2 result on stack.
              MoveMultiple   (SP)+,R0– R1     Restore registers R0 and R1.
              Move           (SP)+,FP         Restore frame pointer registe
              Return                          Return to Subroutine 1.
```

Stack frame



Instructions

- So far we have used
  - Move, Load, Store, Clear, Add, Subtract, Increment, Decrement, Branch, Testbit, Compare, Call, Return
  - Redundant
    - Load and Store could be handled by Move
    - Increment, Decrement could be handled by Add, Subtract
    - Clear same as Move #0,R0
- 8 instructions would have been sufficient, viz. Move, Add, Subtract, Branch, Testbit, Compare, Call, Return
- We actually need a few more

Logic Instructions

- Bitwise operations AND, OR, NOT
- Not
  - Flips all the bits
  - How can I do a 2's complement operation on R1? Many computers have this operation (Negate R1)
- And, Or
  - How can we use And to see whether the leftmost character in R0 (which is 32-bit) is a 'Z' (ascii 01011010)?
  - How can we use Or to set the lsb of R0?

And/Or Examples

And #$FF000000,R0

Compare #$5A000000,R0

Branch=0 YES

Or #1,R0

Logical and Arithmetic Shifts

C ← R0 ← 0

before: 0 | 0 1 1 1 0 · · · 0 1 1

after: 1 | 1 1 0 · · · 0 1 1 0 0

(a) Logical shift left LShiftL #2,R0

0 → R0 → C

before: 0 1 1 1 0 · · · 0 1 1 | 0

after: 0 0 0 1 1 1 0 . . . 0 | 1

(b) Logical shift right LShiftR #2,R0

R0 → C

before: 1 0 0 1 1 · · · 0 1 0 | 0

after: 1 1 1 0 0 1 1 · · · 0 | 1

(c) Arithmetic shift right AShiftR #2,R0

Digit Packing Example

- Two decimal digits are located at LOC and LOC+1
- Wish to represent each as a 4-bit BCD code

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Digit packing code

```
Move      #LOC,R0     R0 points to data.
MoveByte  (R0)+,R1    Load first byte into R1.
LShiftL   #4,R1       Shift left by 4 bit positions.
MoveByte  (R0),R2     Load second byte into R2.
And       #$F,R2      Eliminate high-order bits.
Or        R1,R2       Concatenate the BCD digits.
MoveByte  R2,PACKED   Store the result.
```

Rotate Instructions

C ← R0 (rotate left)
before: C = 0, R0 = 0 1 1 1 0 · · · 0 1 1
after: C = 1, R0 = 1 1 0 · · · 0 1 1 0 1

(a) Rotate left without carry — RotateL #2,R0

R0 → C (rotate right)
before: R0 = 0 1 1 1 0 · · · 0 1 1, C = 0
after: R0 = 1 1 0 1 1 1 0 · · · 0, C = 1

(c) Rotate right without carry — RotateR #2,R0

C ← R0 (rotate left through carry)
before: C = 0, R0 = 0 1 1 1 0 · · · 0 1 1
after: C = 1, R0 = 1 1 0 · · · 0 1 1 0 0

(b) Rotate left with carry — RotateLC #2,R0

R0 → C (rotate right through carry)
before: R0 = 0 1 1 1 0 · · · 0 1 1, C = 0
after: R0 = 1 0 0 1 1 1 0 · · · 0, C = 1

(d) Rotate right with carry — RotateRC #2,R0

Multiplication and Division

- Multiply Ri,Rj (Rj←[Ri]×R[j])
  - Product is 64-bits so most processors store result in Rj (LSW) and R(j+1) (MSW)
- Divide Ri,Rj (Rj←[Rj]/R[i])
  - Quotient in Rj and remainder in R(j+1)
- Some machines have multiply-add (MultiplyAccumulate)
- Not all machines have these instructions

Example: dot product

```
        Move        #AVEC,R1      R1 points to vector A.
        Move        #BVEC,R2      R2 points to vector B.
        Move        N,R3          R3 serves as a counter.
        Clear       R0            R0 accumulates the dot product.
LOOP    Move        (R1)+,R4      Compute the product of
        Multiply    (R2)+,R4        next components.
        Add         R4,R0         Add to previous sum.
        Decrement   R3            Decrement the counter.
        Branch>0    LOOP          Loop again if not done.
        Move        R0,DOTPROD    Store dot product in memory.
```

$$Dotprod = \sum_{i=0}^{n-1} A(i) \times B(i)$$

Sorting

```
for  (j = n−1; j > 0; j = j − 1)
     { for ( k = j−1; k >= 0; k = k − 1 )
         { if   (LIST[k] > LIST[j])
                { TEMP = LIST[k];
                  LIST[k] = LIST[j];
                  LIST[j] = TEMP;
                }
         }
     }
```

(a) C-language program for sorting

```
        Move         #LIST,R0      Load LIST into base register R0.
        Move         N,R1          Initialize outer loop index
        Subtract     #1,R1           register R1 to j = n − 1.
OUTER   Move         R1,R2         Initialize inner loop index
        Subtract     #1,R2           register R2 to k = j − 1.
        MoveByte     (R0,R1),R3    Load LIST(j) into R3, which holds
                                     current maximum in sublist.
INNER   CompareByte  R3,(R0,R2)    If LIST(k) ≤ [R3],
        Branch≤0     NEXT            do not exhange.
        MoveByte     (R0,R2),R4    Otherwise, exchange LIST(k)
        MoveByte     R3,(R0,R2)      with LIST(j) and load
        MoveByte     R4,(R0,R1)      new maximum into R3.
        MoveByte     R4,R3           Register R4 serves as TEMP.
NEXT    Decrement    R2            Decrement index registers R2 and
        Branch≥0     INNER           R1, which also serve
        Decrement    R1              as loop counters, and branch
        Branch>0     OUTER           back if loops not finished.
```

(b) Assembly language program for sorting

Linked list

- Ordered list of items, easy insertion and deletion
- Example, test scores in order of student ID
- Could maintain as array in contiguous memory but what if we want to insert or delete?
- Linked list has a 1-word link field to give address of next entry

Link address
Record 1 → Record 2 → · · · → Record k | 0
Head ... Tail

(a) Linking structure

Record 1 → New record → Record 2 → · · ·

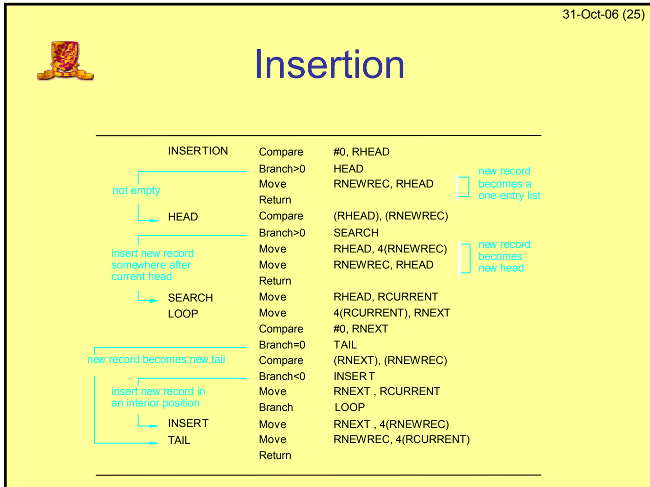
(b) Inserting a new record between Record 1 and Record 2

Student scores in mem

	Memory address	Key field (ID) 1 word	Link field 1 word	Data field (Test scores) 3 words			
First record	2320	27243	1040				Head
Second record	1040	28106	1200				
Third record	1200	28370	2880				
⋮							
Second last record	2720	40632	1280				
Last record	1280	47871	0				Tail

Figure 2.36. A list of student test scores organized as a linked list in memory.

Insertion



Deletion

- Find ID of record to be deleted
- Fix up the link fields

```
DELETION  Compare   (RHEAD), RIDNUM
          Branch>0  SEARCH
          Move      4(RHEAD), RHEAD
          Return
SEARCH    Move      RHEAD, RCURRENT
LOOP      Move      4(RCURRENT), RNEXT
          Compare   (RNEXT), RIDNUM
          Branch=0  DELETE
          Move      RNEXT, RCURRENT
          Branch    LOOP
DELETE    Move      4(RNEXT), RTEMP
          Move      RTEMP, 4(RCURRENT)
          Return
```

not the head record

Machine Instruction Encoding

- Our instructions use different size operands
  - e.g. 32 and 8-bit numbers, 8-bit ASCII characters
  - Need to encode these as a binary pattern in instruction
  - Suppose opcode needs 8-bits
  - e.g. Add R1,R2 needs to specify src and dest registers plus the opcode. Suppose we have 16 registers, need 4-bits for each. Additional bits needed for addressing modes.
  - e.g. Move 24(R0),R5 need OP code, 2 registers and index value of 24. Suppose 3-bits used for addressing mode, need 6 bits in total for addressing mode, 8 bits for registers, 8 bits for opcode. Thus 10 bits are left for the index value.

Encoding

- e.g. Branch>0 LOOP, 8 bits for opcode, 24 bits left for branch address (target address must be within 2^{23} bytes of the branch instruction). Can branch outside this range using an Absolute or register indirect addressing mode (usually called jump instructions)

Encoding

- One possible encoding scheme
- Index value or immediate operand stored in “Other info”
- What about absolute e.g. Move R2, Loc? 18 bits for opcode, addressing mode and register leaving 14 bits which is not enough! Use (b). Can also accommodate “And $FF000000,R2”

8	7	7	10
OP code	Source	Dest	Other info

(a) One-word instruction

OP code	Source	Dest	Other info
Memory address/Immediate operand			

(b) Two-word instruction

OP code	*Ri*	*Rj*	*Rk*	Other info

(c) Three-operand instruction

CISC and RISC

- If we allow Move LOC1,LOC2 need three words!
- Multiple length instructions are difficult to implement with high clock rate
- Complex instruction set computers (CISC) have complex instruction encodings like this (e.g. IA-32)
- Reduced instruction set computers (RISC) only allow simple 32-bit formats, few addressing modes and all data to be manipulated must be in registers e.g. Add (R3),R2 is not allowed, instead use Move (R3),R1 followed by Add R1,R2 (e.g. ARM)
  - RISC machines often are 3-address machines as the addressing mode field is either not necessary or simplified e.g. Add R1,R2,R3
- CISC machines usually require less instructions but have a lower clock rate, RISC require more instructions but have a higher clock rate. Very detailed tradeoff analysis is required to find best solution
  - Backward compatibility may also be an issue