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?

Passing via registers

- One way is putting the parameters in registers

  Calling program
  - Move N, R1
  - Move #NUM1, R2
  - Call LISTADD
  - Move R0, SUM

  Subroutine
  - LISTADD
  - Clear R0
  - Initialize sum to 0.
  - LOOP
  - Add (R2)+, R0
  - Decrement R1
  - Branch>0 LOOP
  - Return

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?

Parameters on Stack

- A stack can handle a large (and variable) number of parameters

  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

Example

Main program:

- Move PARAM2, (SP)
- Place parameters on stack.
- Move PARAM1, (SP)
- Call SUB1
- Move (SP),RESULT
- Store result.
- Add #8,SP
- Restore stack level.
- Next instruction.

First subroutine:

- Move FP, (SP)
- Save frame pointer register.
- Move SP,FP
- Load the frame pointer.
- Move Multiple R0 R3, (SP)
- Save registers.
- Move 8(FP),R0
- Get first parameter.
- Move 12(FP),R1
- Get second parameter.
- Move PARAM3, (SP)
- Place a parameter on stack.
- Call SUB2
- Move (SP)+,R2
- Pop SUB2 result into R2.
- Move Multiple (SP)+,R0 R3
- Restore registers.
- Move (SP)+,FP
- Restore frame pointer register.
- Return.
- Return to Main program.

Second subroutine:

- Move FP, (SP)
- Save frame pointer register.
- Move SP,FP
- Load the frame pointer.
- Move Multiple R0 R1, (SP)
- Save registers R0 and R1.
- Move 8(FP),R0
- Get the parameter.
- Move R1,8(FP)
- Place SUB2 result on stack.
- Move Multiple (SP)+,R0 R1
- Restore registers R0 and R1.
- Move (SP)+,FP
- Restore frame pointer register.
- Return.
- Return to Subroutine 1.
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 Isb of R0?

And/Or Examples

And #$FF000000, R0
Compare #$5A000000, R0
Branch=0 YES

Or #1, R0

Logical and Arithmetic Shifts

Logical shift right LShiftR #2, R0
Logical shift left LShiftL #2, R0

Logical shift right LShiftR #2, R0
Logical shift left LShiftL #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

<table>
<thead>
<tr>
<th>Decimal digit</th>
<th>BCD code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
</tbody>
</table>

Digit Packing code

Move #LOC,R0
MoveByte (R0)+, R1
LShiftL #4, R1
MoveByte (R0), R2
And #$F, R2
Or R1, R2
Move R2, PACKED

R0 points to data.
Load first byte into R1.
Shift left by 4 bit positions.
Load second byte into R2.
Eliminate high-order bits.
Concatenate the BCD digits.
Store the result.
Rotate Instructions

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

before: \[ \ldots 000111 \ldots \] 
after: \[ \ldots 110010 \ldots \]

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

before: \[ \ldots 000111 \ldots \] 
after: \[ \ldots 001010 \ldots \]

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

before: \[ \ldots 000111 \ldots \] 
after: \[ \ldots 101100 \ldots \]

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

before: \[ \ldots 000111 \ldots \] 
after: \[ \ldots 001010 \ldots \]

Multiplication and Division

• Multiply Ri, Rj \( (Rj \rightarrow [Ri] \times [Rj]) \)
  – Product is 64-bits so most processors store result in Rj (LSW) and R(j+1) (MSW)
• Divide Ri, Rj \( (Rj \leftarrow \lfloor \frac{[Ri]}{[Rj]} \rfloor) \)
  – Quotient in Rj and remainder in R(j+1)
• Some machines have multiply-add (MultiplyAccumulate)
• Not all machines have these instructions

Example: dot product

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

Sorting

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

\[
\text{b) Assembly language program for sorting}
\]

\[
\begin{align*}
\text{for } (j = n 1; j > 0; j = j 1) & \\
\text{for } (k = j 1; k > =0; k = k 1) & \\
\text{if } \text{LIST}(k) > \text{LIST}(j) & \\
\text{TEMP} = \text{LIST}(k); & \\
\text{LIST}(k) = \text{LIST}(j); & \\
\text{LIST}(j) = \text{TEMP}; & \\
\end{align*}
\]

\[
\text{a) C-language program for sorting}
\]

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

\[
\begin{align*}
\text{Record 1} & \quad \text{Linking structure} \\
\text{Record 1} & \quad \text{New record} \\
\end{align*}
\]

Student scores in mem

\[
\begin{align*}
\text{Record 1} & \quad \text{Record 2} & \quad \text{Records} & \quad \text{0} \\
\text{Head} & \quad \text{Data} & \quad \text{Link} & \\
\text{Record 1} & \quad \text{Record 2} & \quad \text{NEW RECORD} & \quad \text{NEW RECORD} \\
\end{align*}
\]

\[
\begin{align*}
\text{Figure 2.36. A list of student test scores organized as a linked list in memory.}
\end{align*}
\]
**Insertion**

- Compare RHEAD, RHEAD
  - Branch>0: Move RNEWREC, RHEAD
  - Return
- Compare (RHEAD), (RNEWREC)
  - Branch>0: SEARCH
  - Move RHEAD, 4(RNEWREC)
- Move RNEWREC, RHEAD
  - Return
- Move RHEAD, RCURRENT
- Move 4(RCURRENT), RNEXT
  - Compare
  - Branch=0: TAIL
  - Branch<0: #0, RNEXT

**Deletion**

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

**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)

**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