CSC2510 - Computer Organization

Lecture 4: Machine Instructions and Programs

Philip Leong

Indexed Addressing

- **Index**
  - \(X(Ri)\)
  - \(EA = [Ri] + X\)

- **Base with index**
  - \((Ri,Rj)\)
  - \(EA = [Ri] + [Rj]\)

- **Base with index and offset**
  - \(X(Ri,Rj)\)
  - \(EA = [Ri] + [Rj] + X\)

Indexed Addressing Variations

Relative Addressing

- **Relative (offset from the PC)**
  - Recall PC determines the address of the next instruction to execute
  - \(X(\text{PC})\)
  - \(EA = [\text{PC}] + X\)

- Used mainly for loops e.g., below, use relative address to get the update the PC. What is \(X\)?

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

Assembly Language

- **Assembly language**
  - Machine code

- We use mnemonics to express an assembly language in a symbolic manner

- Assembly language is a 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

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>AMODE1</th>
<th>AMODE2</th>
<th>OPERAND1</th>
<th>OPERAND2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>REG</td>
<td>ABS</td>
<td>0</td>
<td>SUM</td>
</tr>
</tbody>
</table>

#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
  – 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

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?

Assembly language vs machine code

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 Test bit #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 Test bit #3, INSTATUS Wait for a character to be entered in the keyboard buffer DATAIN.
Branch=0 READ in the keyboard buffer DATAIN into R1.

ECHO Test bit #5, OUTSTATUS Wait for the display to become ready.
Branch=0 ECHO
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. If it is not CR, then branch back and read another character.

Move #LOC, R0 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
Add #4, SP
Move (SP), ITEM ; pop
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?

Safe pop/push

SAFEPOP
Compare Branch=0
#2000, SP
EMPTYERROR
Move (SP), ITEM Otherwise, pop the top of the stack into memory location ITEM.

SAFE PUSH
Compare Branch=0
#1500, SP
FULLERROR
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
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.

Subroutine call

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.

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?

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.