Programming in Assembler

Need knowledge of CPU

8051 Programmers model

• what registers are available?
• what memory is available?
  • code memory (for programs)
  • data memory (for variables and the stack)
• what instructions are available

Programming in assembler involves mapping problem onto these resources

• select instructions that will have desired effect
• not all desired instructions may be available
• not all addressing modes may be available

Assembler Programming

• not so much difficult as detailed
• complexity is caused by the multiplicity of choice
• you have to worry about much more than with HLLs

Familiarity with the Instruction Set is Critical!

What Registers are available?

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acc</td>
<td>7-0</td>
<td>All arithmetic, comparisons, logic</td>
</tr>
<tr>
<td>B</td>
<td>7-0</td>
<td>temporary, used for Multiply &amp; Divide</td>
</tr>
<tr>
<td>R0</td>
<td>7-0</td>
<td>scratchpad, can be used as pointer</td>
</tr>
<tr>
<td>R1</td>
<td>7-0</td>
<td>scratchpad, can be used as pointer</td>
</tr>
<tr>
<td>R2</td>
<td>7-0</td>
<td>scratchpad</td>
</tr>
<tr>
<td>R3</td>
<td>7-0</td>
<td>scratchpad</td>
</tr>
<tr>
<td>R4</td>
<td>7-0</td>
<td>scratchpad</td>
</tr>
<tr>
<td>R5</td>
<td>7-0</td>
<td>scratchpad</td>
</tr>
<tr>
<td>R6</td>
<td>7-0</td>
<td>scratchpad</td>
</tr>
<tr>
<td>R7</td>
<td>7-0</td>
<td>scratchpad</td>
</tr>
<tr>
<td>Sp</td>
<td>7-0</td>
<td>Stack pointer to Internal data memory</td>
</tr>
<tr>
<td>PSW</td>
<td>7-0</td>
<td>Processor Status Word</td>
</tr>
<tr>
<td>D PTR</td>
<td>15-0</td>
<td>Data Pointer Register, only 16 bit register, used as pointer to code memory</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>Carry bit, used for multi-byte add/sub</td>
</tr>
</tbody>
</table>

Constraints

Accumulator (Acc – the A register) must be used for arithmetic & logic operations:

add a,#100 is OK (acc:= acc + 100)
add r1,#100 is NOT (can’t add to r1)

- only r0 & r1 can be used as pointers to internal memory
- only B can be used as second argument for MUL & DIV
- r0 - r7, are the same as Memory locations 0 to 7

Assembler Programming - like early Basic

The early versions of the Basic programming language had the following constructs:

• LET variable = expression
• GOTO line #
• IF condition GOTO line #
• INPUT variable
• PRINT variable | string
• GOSUB line #
• RETURN

Where:

<condition> ::= <expression> <rel-op> <expression>
<rel-op> ::= < | = | = | # | >= | >
<expression> ::= <value> <operator> <value>
<value> ::= variable | number
<operator> ::= + | - | * | /

What’s Missing?

No support for subroutines with parameters, while/repeat loops, grouped statements.

• these higher level constructs must be synthesised.

Assembler Data Movement Instructions

Let write an assembler program to mimic:

\[ \text{var } X, Y : \text{byte} ; \]
\[ Y := Y + X ; \]

Decide on Memory Locations for Variables

• variables we need to store them somewhere
• their actual memory location doesn't matter …
• 8051 only has 128 bytes of memory

Arbitrarily, let’s use location 20 for X and 30 for Y

mov A,30 ; Move Y to the accumulator
adda 20 ; Acc’ now contains Y+X
mov 30,A ; Put total back into Y

Notice:

• the MOV instruction move Right to Left
• the arguments are simple memory addresses

Using Symbolic Names

This is hard to read, So define constants for the memory adr

\[ X \text{ equ } 20 \]
\[ Y \text{ equ } 30 \]

mov a,Y ; Move Y to the accumulator
adda X ; Acc’ now contains Y+X
mov Y,a ; Put total back into Y
Assembler Flow-of-Control Instructions

Execution order of instructions

Instructions are normally executed in order

- the Program Counter (PC) is incremented after each instruction is fetched.

Altering the execution order can be done:

- using an UNCONDITIONAL JUMP (sjmp/ljmp)
  - these simply load a new value into the PC

```
ljmp 200 ; Set PC to 200
```

- using a CONDITIONAL JUMP
  - these test a value and depending on the result, MAY ALTER the PC
  - if condition fails, the conditional jump has no effect

```
jz 200 ; JumpIf Zero to 200
    ; Jump to 200 if Acc = Zero
```

There are many different conditional jumps (sometimes called Branches)
eg, jump-if-zero, jump-if-NOT-zero, jump-if-carry-set …

General form of Assembler Source

```
; Fill locations 50-70h with zero
ORG 20h
Start:  mov sp,#127 ; Setup Stack
    mov r1,#50h
    mov A,#0
lup:   mov @r1,A   ; mem[r1] = 0
    inc r1    ; r0++
cjne r1,#71h,lup ; again?
Wait:  sjmp wait   ; Hang here
```

Points to Note

- labels start in column 1
- instructions NEVER start in Column 1
- instructions have varying number of parameters
- ; indicates a comment – remainder of line is ignored
- some number are hexadecimal
  - if followed by h (eg 30h) or preceded with $ (eg $30)
- ORG defines where next byte is placed in memory but doesn’t generate any code

EQU is a constant definition (can’t be redefined)

```
zero  EQU  0
MOV  A,#zero
```

Well Commented Assembler

```
; PUTCH() - Send the character in acc to the Serial Port
;---------------------------------------------------------------------
putch:  clr      ti               ; Clear “Char Been Sent” Bit
    mov sbuf,a     ; Store character to Output
    gone:  jnb ti,gone   ; Wait until it’s gone
    ret   ; return
;********************************************************************************
; NEWLINE() - send a RETURN & LINEFEED to the display ;
newline: mov a,#13  ; RETURN character in Acc
    lcall   putch
mov a,#10  ; Linefeed character
    lcall putch
    ret
;********************************************************************************
; PRINT a string ;------------------------------------------------------------------------------
; Print a string ending in a zero byte to the display (serial port). ;
; Parameters     :  r1 - the address of the first byte of the string
; Return Values  : none
; Regs changed    : acc, R1
;------------------------------------------------------------------------------
repeat
print:  mov a,@r1  ;   acc = *r1    // Get a byte
    jz done  ;   if acc == 0 then break;
lcall putch ;   putch(acc) - Send it to serial port
    inc r1   ;   r1:= r1+1   // point r1 at next char
    sjmp print ;   until false   // do next char
    done:  ret
```

Input Syntax (Structure)

A grammar is needed to discuss the form (the structure) of a language, such as assembler.

This is itself a language being used to describe a language.

The grammar rules below are modelled on Extended BNF (Backus-Naur Form):

- a Grammar consists of a series of Productions
- each Production defines a possible combination of Productions and Terminal Symbols
- Productions are surrounded by < > and are defined elsewhere.
- Terminal Symbols indicate actual symbols/characters.

The following symbols have a special meaning:

- <XXX> this a placeholder for a production called XXX which is defined elsewhere.
- Whatever matches the production <XXX> maybe inserted instead of <XXX>.
- [ OBJ ] OBJ is optional – it may occur 0 or one time
- { OBJ } OBJ may be present zero or more times
- ::= is read as “is defined as”
- | means OR

Recursion in the Productions is permitted
Lexical and Syntactic Structure

The following rules specify structure of an assembler source file but don’t specify the exact makeup of the symbols that may be used for identifiers, operators inside expressions.

The definition of the actual legal (permitted symbols) are defined elsewhere is the Lexical Grammar.

\[
\text{<Assembler file> ::= } \{ \text{<assembler line>} \}
\]

\[
\text{<assembler line> ::= [ <label> ] [<mnemonic> [<arg-list>]] [<comment>]}
\]

\[
\text{<label> ::= <identifier>}
\]

\[
\text{<mnemonic> ::= one of defined opcodes}
\]

\[
\text{<arg-list> ::= <argument> {"," <argument>}}
\]

\[
\text{<argument> ::= #<expression> | <expression> | Rn | A}
\]

\[
\text{<expression> ::= something returning a numeric value}
\]

\[
\text{<comment> ::= ";" rest of the line}
\]

Assembler Programming  9 Copyright 2006 – Giovanni Moretti

Assembler vs High Level Languages

<table>
<thead>
<tr>
<th>High Level Languages</th>
<th>Assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction is easy</td>
<td>No Abstractions - initially</td>
</tr>
<tr>
<td>Multiple Data types</td>
<td>Only those supported on underlying Hardware</td>
</tr>
<tr>
<td>char, int, double, string</td>
<td></td>
</tr>
<tr>
<td>Structured data types</td>
<td>None – must be synthesized – usually with pointers</td>
</tr>
<tr>
<td>(arrays, structs)</td>
<td></td>
</tr>
<tr>
<td>Algorithmic Constructs</td>
<td>None, only:</td>
</tr>
<tr>
<td>IF - THEN – ELSE</td>
<td>• JUMP</td>
</tr>
<tr>
<td>WHILE - DO …</td>
<td>• Conditional Jumps</td>
</tr>
<tr>
<td>Switch statements</td>
<td></td>
</tr>
<tr>
<td>Functions with parameters/return values</td>
<td>Only Subroutine calls</td>
</tr>
<tr>
<td>- no parameters</td>
<td></td>
</tr>
<tr>
<td>- no return values</td>
<td></td>
</tr>
<tr>
<td>Object-oriented programming</td>
<td>---</td>
</tr>
<tr>
<td>Not machine-specific – can move (relatively) easily</td>
<td>Move to different architecture is very time-consuming</td>
</tr>
</tbody>
</table>

Control & Responsibility

No control over allocation of variables
No control over assembler constructs used
Programmer is responsible for all variable allocation
Programmer must select specific instructions

Summary

Good for rapid development of larger systems when program size and execution speed aren’t constraints
• Good when max’ speed or min’ size is required
• Not good for large systems -too much detail

Assembler Overview

Processors need:
- instructions
- data

Both of these are located at a specific address in memory
- instructions are decoded from bit patterns (patterns of bits have predefined meaning)
- meaning of the bits is arbitrary but defined by the CPU designer

To generate an instruction, three pieces of information are needed:
- the actual instruction to be executed
- the values of any arguments
- where in memory to place the instruction

All of these must be made available to the assembler

Differing types of Operands

The data being moved by a MOV opcode may be:
- a constant value
- currently in a memory location
- in a memory location whose address can be calculated
- in a named register (eg A (accumulator) B, r0-r7, SP which must be indicated unambiguously …

Addressing Mode Indicators

The differing types of operands must be uniquely identifiable as:
- variables aren't typed
- a variable can be either data or an address
- differing arguments are encoded differently
  - mov a,r1 – instruction can be one byte long
  - mov a,217 – needs an extra byte for the address

Indicating the Types of Arguments

The different types of arguments must be distinctive.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>the Accumulator register</td>
</tr>
<tr>
<td>Rn</td>
<td>one of R0 – R7</td>
</tr>
<tr>
<td>a number</td>
<td>Direct - Internal Memory 8 bit address</td>
</tr>
<tr>
<td>@Ri</td>
<td>Internal ram 0-255 via R0/ R1</td>
</tr>
<tr>
<td>#data</td>
<td>a 8 bit value</td>
</tr>
<tr>
<td>#data16</td>
<td>a 16 bit value</td>
</tr>
<tr>
<td>relative</td>
<td>signed offset (one byte) –128 to +127</td>
</tr>
<tr>
<td>bit</td>
<td>Directly addressable bit in Data Memory</td>
</tr>
<tr>
<td>addr11</td>
<td>11 bit destination ( acall &amp; ajmp )</td>
</tr>
<tr>
<td>addr16</td>
<td>16 bit destination (lcall &amp; ljmp) can be any byte in 64K program memory</td>
</tr>
</tbody>
</table>
Assembler Instructions

**LAYOUT MATTERS** - instructions are written one per line
e.g.  MOV <dest-byte>,<src-byte>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>30h</td>
<td>47h or $47</td>
</tr>
<tr>
<td>47h</td>
<td>15h or $15</td>
</tr>
</tbody>
</table>

After Instruction Completes

- mov  R0, 30  ; R0 = 47h
- mov  R0,#30h ; R0 = 30h
- mov  A, @r0  ; A = 47h
- mov  $47, #255 ; $47 = 255
- mov  $47, #$FF ; $47 = 255

**Parameter Types**

- **value**
  - Direct address
    - value is memory address
    - mov 30,40 ; mem[30] = mem[40]
  - #value
    - Immediate data value is used (no reference to mem)
    - mov 30,#0 ; mem[30] = 0
  - R0 or A
    - Register is used as source/destination
    - mov r0,#15
  - @R0
    - Indirect contents of register are used as addr
    - mov @r0,'A' ; mem[15] = 'A'

What’s needed to Assemble a Program?

=> A list of *all the possible* instructions variants

This is NOT a list like:  mov, sjmp, cjne, inc…

but a list that showing:

- each possible address
- data type variation (bit, byte, 16 bits)
- applicable opcode

- mov direct,A
  - 1st byte  2nd byte  3rd byte
  - 1111 0101  direct address

- mov direct,direct
  - 1st byte  2nd byte  3rd byte
  - 1000 0101    Direct-adr (src)     direct-adr (dest)

- mov direct,@Ri
  - 1st byte  2nd byte  3rd byte
  - 1000 0111    direct address

- mov direct,#data
  - 1st byte  2nd byte  3rd byte
  - 0111 0101    direct address     data-value

Eg  mov 100, #22  would be encoded as 75 64 16

Building Instructions – e.g. MOV opcode

Opcodes need to encode all the information for one instruction, such as:

- which Operation (mov, incr, jmp …)
- arguments (address of source and/or destination)
- any constant value

The Size of the Instructions Vary

Instructions are designed to be as compact as possible

Why?

- mov direct,direct - at least three bytes long – why?

  - Rn  is one of R0 - R7
  - Ri  is one of R0 or R1
  - dir  is a direct (one byte) address

- F4  mov, A, Rn,  1 1110 1RRR
- F5  mov, A, direct, 2 1110 0101,dir
- F6  mov, A, @Ri,    1 1110 0111

 Opcode Size

The sequence of bytes for one opcode may be a merging of an opcode template with information from the arguments:

eg  RRR is replaced with 000-111 representing R0-R7,
    i    is replaced with 0 or 1 representing R0 or R1

Hardware-supported Instructions

**Data Manipulation Instructions**

- Move primitive data types - bits, bytes, words
  - move reg-to-reg, reg to/from memory
- Clear Registers or memory
- Set (make all ones) Registers or memory
- Increment & Decrement Regs/memory

**Arithmetic and Logical Instructions**

These work on bytes, words (& sometimes bit)

- Add, subtract, sometimes multiply, less frequently
- AND, OR, NOT (called Complement), XOR

**Flow of Control Instructions**

- JUMP – like a GOTO
- JUMP if a condition is TRUE
- Call a subroutine (& remember how to get back)
- Looping Control
  - Fixed count - Decrement & Jump if not zero DJNZ
  - Compare and Jump if not equal CJNE
**Hardware-supported Instructions**

**Status Manipulation Instructions**
- Allow / disallow interrupts
- Set up Stack Pointer
- Comparisons on primitive data types & set flags
  - Negative, Zero, Carry

**Advanced Features**
- String support – move, comparisons
- Floating point arithmetic

**These aren't supported at all**
- if the processor is to work on more complex data types, (eg strings or floating point numbers), subroutines must be written to implement the operations.

**Programmer’s Model – Detail**

The processor has several sets of:
- Programmer registers
- I/O control registers
- Status registers
- I/O ports

ALL of which as accessed as though they were memory

**Many memory locations have predefined alternate uses**

**Programmer-accessible Registers**

**Accumulator – Acc** at E0H
This is used for:
- Arithmetic ops - ADD, ADDC, SUBB, MUL, DIV
- logical operations - ANL, ORL, XRL, CPL
- Rotates - RL, RLC, RR, RRC
- SWAP - Swap top/lower four bits

**B Register – at F0H**
The is an auxiliary register used primarily in MUL & DIV instructions. Can be used at temp location otherwise

**Register Bank – R0 to R7**
These are (usually) the first 8 locations in internal memory (0H – 7H):
- Treated as special – only need 3 bits to specify them
- Many instructions have one of R0-R7 as source or dest

**R0 and R1 are special**
- Have an INDIRECT addressing mode
- can be used as pointer registers

**DPTR – Data Pointer Register**
- the only 16 bit register
- used as pointer to CODE memory
- can be accessed as two 8 bit regs – DPL & DPH

**Processor Stack**
The 8051 supports a stack by having a register (SP)

- that auto-increments when used in a PUSH instruction
- auto-decrements when used in a POP instruction

**Where’s the Stack Stored?**
The stack itself is stored in data memory wherever SP location happens to be pointing

⇒ **make sure that the memory used for the stack is free!**

**Stack Pointer – SP** at 81H
A processor manipulated register used to point to last byte stacked – it’s incremented BEFORE a new byte is stacked.

**Stack-oriented Instructions**

**PUSH Acc**
Push the acc onto the stack

\[
\text{SP} = \text{SP} + 1 \\
\text{Data Mem}(\text{SP}) = \text{Acc}
\]

**POP Acc**
Pop the top-of-stack into Acc

\[
\text{Acc} = \text{Data Mem}[\text{SP}] \\
\text{SP} = \text{SP} - 1
\]
Support for Subroutine Calls

The stack is used to store the return address during a subroutine call.

- before jumping to a subroutine the address to return to is stacked (done by LCALL)
- at the end of the subroutine, the return opcode (RET) pops this address and puts it back into PC

**LCALL sub-adr**  Call a subroutine

- PC = PC + 3 ; point at next instr'
- SP = SP + 1
- DataMem[SP] = PC (bits 0-7) ; stack LO part of PC
- SP = SP + 1
- DataMem[SP] = PC (bits 8-15) ; stack HI part of PC
- PC = sub-adr

**RET**  Return from a subroutine

- causes the address to return to be popped from stack into the program counter

- PC(bit 8-15) = DataMem[SP] ; Pop return adr HI
- SP = SP – 1 ; cut stack back
- PC(bit 0-7) = DataMem[SP] ; Pop return adr LO
- SP = SP – 1 ; cut stack back

Status Register – PSW – at D0H

The Processor Status Word register (PSW) is a register that holds:

- status information from the result of previous instructions (eg did a carry occur?)
- some configuration bits (RS1 & RS0) – see next slide

<table>
<thead>
<tr>
<th>CY</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>PSW1</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carry Bit</td>
<td>Aux Carry</td>
<td>Flag -</td>
<td>Register Bank Select</td>
<td>Overflow</td>
<td>Spare</td>
<td>Parity</td>
<td></td>
</tr>
<tr>
<td>Set if a carry from Acc.7</td>
<td>free for user</td>
<td>0=&gt; RB0</td>
<td>1=&gt; RB1</td>
<td>2=&gt; RB2</td>
<td>3=&gt; RB3</td>
<td>Set by arithmetic Ops if SIGNED Integer Overflow</td>
<td>free for user</td>
</tr>
</tbody>
</table>

Register Bank Location

By default, R0 is located in memory location 0, R1 in 1 ... This can be altered by the user to three alternate locations

**RB1 – Register Bank 0**

<table>
<thead>
<tr>
<th>PSW – RS1&amp;0 = 00</th>
<th>PSW – RS1&amp;0 = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 Data RAM 0</td>
<td>R0 Data RAM 10H – 16</td>
</tr>
<tr>
<td>R1 Data RAM 1</td>
<td>R1 Data RAM 11H – 17</td>
</tr>
<tr>
<td>R2 Data RAM 2</td>
<td>R2 Data RAM 12H – 18</td>
</tr>
<tr>
<td>R7 Data RAM 7</td>
<td>R7 Data RAM 17H – 23</td>
</tr>
</tbody>
</table>

**RB1 – Register Bank 1**

<table>
<thead>
<tr>
<th>PSW – RS1&amp;0 = 01</th>
<th>PSW – RS1&amp;0 = 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 Data RAM 8</td>
<td>R0 Data RAM 18H – 24</td>
</tr>
<tr>
<td>R1 Data RAM 9</td>
<td>R1 Data RAM 19H – 25</td>
</tr>
<tr>
<td>R2 Data RAM AH (10)</td>
<td>R2 Data RAM 1AH – 26</td>
</tr>
<tr>
<td>R7 Data RAM FH (15)</td>
<td>R7 Data RAM 1FH – 31</td>
</tr>
</tbody>
</table>

**RB1 – Register Bank 2**

<table>
<thead>
<tr>
<th>PSW – RS1&amp;0 = 02</th>
<th>PSW – RS1&amp;0 = 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 Data RAM 0</td>
<td>R0 Data RAM 10H – 16</td>
</tr>
<tr>
<td>R1 Data RAM 1</td>
<td>R1 Data RAM 11H – 17</td>
</tr>
<tr>
<td>R2 Data RAM 2</td>
<td>R2 Data RAM 12H – 18</td>
</tr>
<tr>
<td>R7 Data RAM 7</td>
<td>R7 Data RAM 17H – 23</td>
</tr>
</tbody>
</table>

Changing Register Banks

Need to alter RS1 and RS0 in the Processor Status Word
e.g. to move to register Bank 3

```
mov acc, psw  Get PSW
orl #18h   merge in RS1 & RS0
mov psw, acc  Put result back
```

OR

```
setb psw.4   Set RS1
setb psw.3   and RS0
```

In this paper:  there will be no need to use alternate register banks but you should be familiar with the underlying idea.
**Special Function Registers**

The registers for controlling the I/O devices are mapped so as to appear as locations in internal (data) memory. They appear between locations 80H and FFH.

### Internal Memory

<table>
<thead>
<tr>
<th>Bank Select</th>
<th>Directly Addressable Bits 0-7F</th>
<th>Reset value of Stack Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>RB0*</td>
<td></td>
</tr>
<tr>
<td>01H</td>
<td>RB1*</td>
<td></td>
</tr>
<tr>
<td>02H</td>
<td>RB2*</td>
<td></td>
</tr>
<tr>
<td>03H</td>
<td>RB3*</td>
<td></td>
</tr>
<tr>
<td>10H</td>
<td>R0H</td>
<td></td>
</tr>
<tr>
<td>11H</td>
<td>R1H</td>
<td></td>
</tr>
<tr>
<td>12H</td>
<td>R2H</td>
<td></td>
</tr>
<tr>
<td>13H</td>
<td>R3H</td>
<td></td>
</tr>
<tr>
<td>14H</td>
<td>R4H</td>
<td></td>
</tr>
<tr>
<td>15H</td>
<td>R5H</td>
<td></td>
</tr>
<tr>
<td>16H</td>
<td>R6H</td>
<td></td>
</tr>
<tr>
<td>17H</td>
<td>R7H</td>
<td></td>
</tr>
</tbody>
</table>

* Four banks of Registers addressable as R0-R7

### Directly Addressable

- Bits 0-7F

### Reset value of Stack Pointer

- 00H

Several common registers are:

- Acc: E0H
- B register: F0H
- PSW: D0H
- IE: E8H
- Port 0: 80H
- Port 1: 90H
- Port 2: A0H
- Port 3: B0H

These registers are **BIT Addressable**:

- you can use SetB,ClrB,JB, JNB, CPL their bits
- e.g: cpl p3.7

### What goes where in Memory?

There are few constraints with assembler programming but some of them are absolute:

- **the first instruction to be executed is that at address ZERO of PROGRAM memory**
  - this is called the Reset Address
  - Your program must start at address zero

- **there is ONLY 128 bytes of Data Memory** for:
  - registers r0 – r7 (usually in locations 0 to 7)
  - the stack
  - any variables you need

Locations 8-127 must be shared between variables and the stack.

The Stack is used by subroutine calls so SP MUST NOT point at data memory that contains variables.

### How much Stack Space should be allocated?

- the primary use of the stack is to store subroutine return addresses
- two bytes per NESTED subroutine call (call from inside a subroutine)

**For this paper, a 16-24 bytes of stack space will be plenty!**

### Sample Memory Layout #1

- few subroutine calls will be made
  - little stack space required (2 bytes/call)
- bit addressable registers not required
- don’t use memory locations 0-7 (used for r0 to r7)

**Data Memory**

```
SP 110
Stack 127

110

110

7

0
```

; Can use locations 8 to 110 for variables
; Define Data memory locations to be used for variables

```
x equ 8
y equ 9
```

Start: mov sp, #110 ; setup Stack at 110
your instructions goes here!
stop sjmp stop ; make program stops
Sample Memory Layout #2

- Few subroutine calls will be made
  => Little stack space required (2 bytes/call)
- Bit addressable registers not required
- Don’t use memory locations 0-7 (used for r0 to r7)

```
org 0   ; Start program at 0
start mov sp, #7  ; Setup Stack at 7
    your instructions go in here!
stop sjmp stop  ; make program stops
```

Variables

Data Memory

; Stack starts at 8, can use up to location 31
; Can use locations 32 to 127 for variables
; Define Data memory locations to be used for variables

```
X equ 32
Y equ 33
```