Lecture - 07
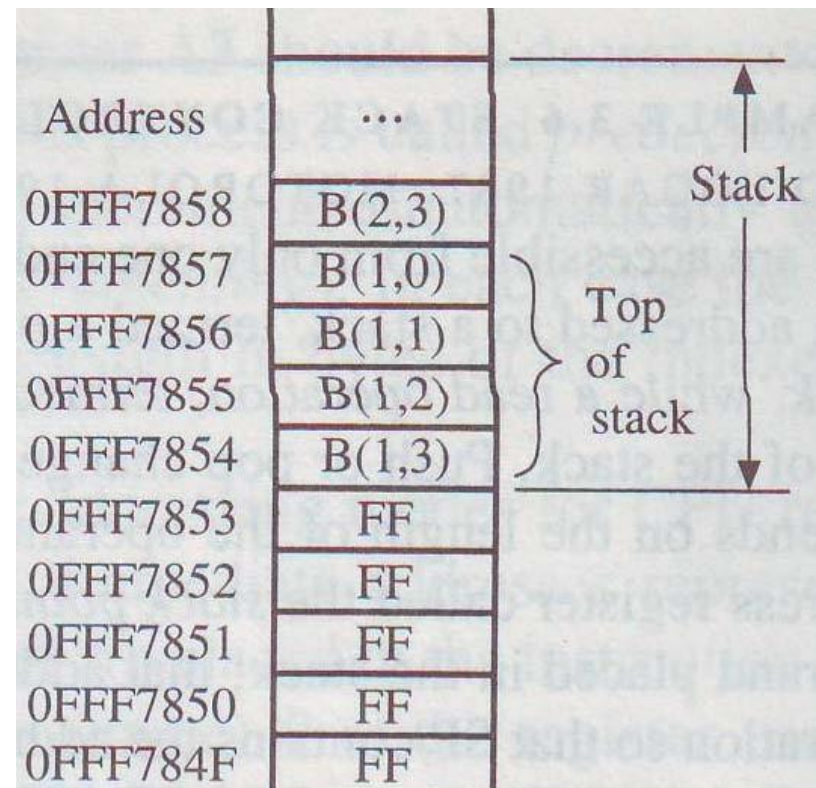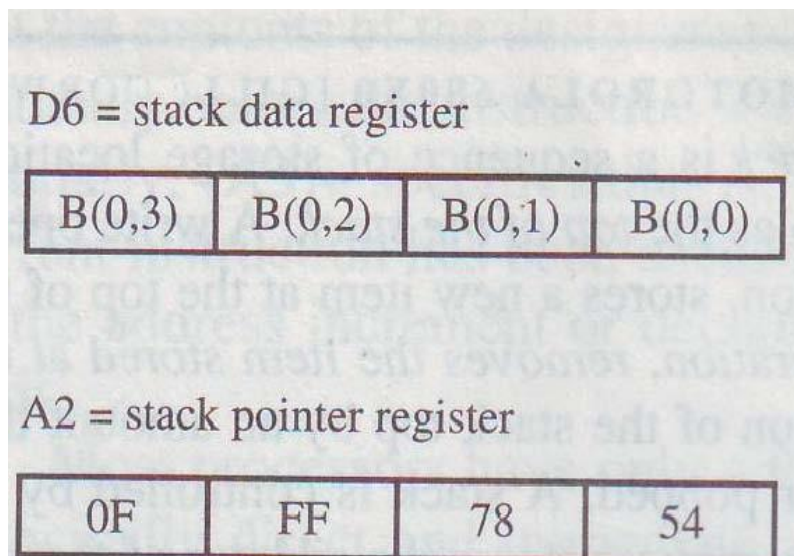
Stack Control

- A *stack* is a sequence of storage locations that are accessible from only one end referred to as the *top of the stack*.

- Two operations: *Push* and *Pop*.

- Push or pop changes the position of the stack top by an amount that depends on the length of the operand pushed or popped.

- A stack is controlled by an address register called the *stack pointer SP*. It contains the address of the new stack top.
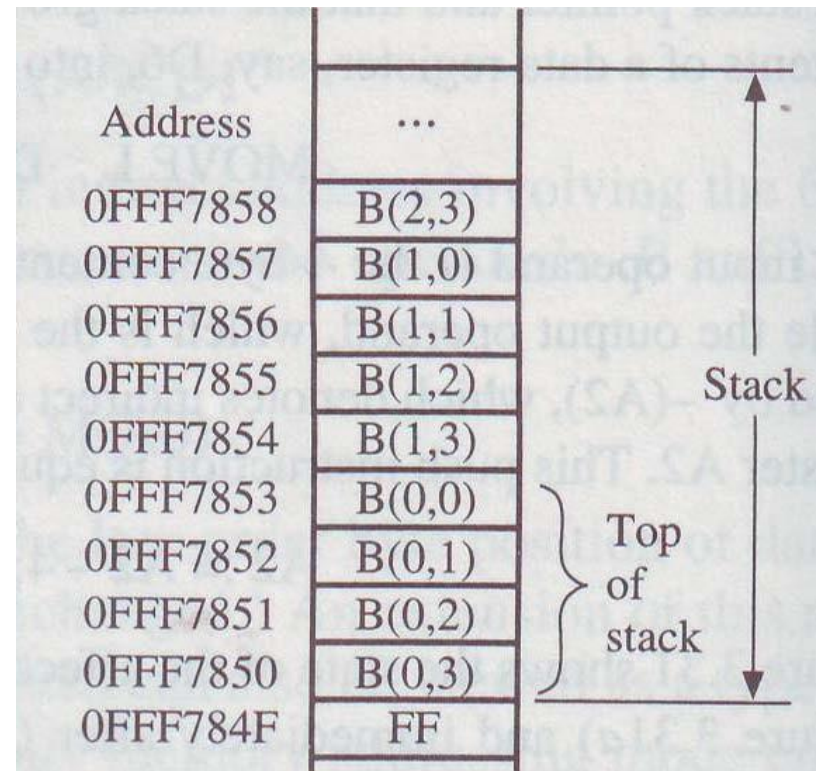
Stack Control in Motorola 680X0

- Motorola 680X0 has no explicit hardware for stack support. But its various addressing modes make it easy to treat any contiguous region of its external memory M as a stack.

- Suppose the address register A2 of the 680X0 is designated as stack pointer and the stack grows toward the low addresses of M.

- To push the content of D6 (data register) into the stack requires a single instruction MOVE.L D6, -(A2) which is equivalent to A2:=A2-4 and M(A2):=D6.

- The pop instruction is MOVE.L (A2)+, D6 which is equivalent to D6:=M(A2) and A2:=A2+4

Stack Control in Motorola 680X0

D6 = stack data register

B(0,3)	B(0,2)	B(0,1)	B(0,0)

A2 = stack pointer register

0F	FF	78	54

Address	...	
		Stack
0FFF7858	B(2,3)	
0FFF7857	B(1,0)	Top
0FFF7856	B(1,1)	of
0FFF7855	B(1,2)	stack
0FFF7854	B(1,3)	
0FFF7853	FF	
0FFF7852	FF	
0FFF7851	FF	
0FFF7850	FF	
0FFF784F	FF	

Prior the execution of MOVE.L D6, -(A2)

Stack Control in Motorola 680X0

D6 = stack data register

B(0,3)	B(0,2)	B(0,1)	B(0,0)

A2 = stack pointer register

0F	FF	78	50

Address	...
0FFF7858	B(2,3)
0FFF7857	B(1,0)
0FFF7856	B(1,1)
0FFF7855	B(1,2)
0FFF7854	B(1,3)
0FFF7853	B(0,0)
0FFF7852	B(0,1)
0FFF7851	B(0,2)
0FFF7850	B(0,3)
0FFF784F	FF

Stack

Top
of
stack
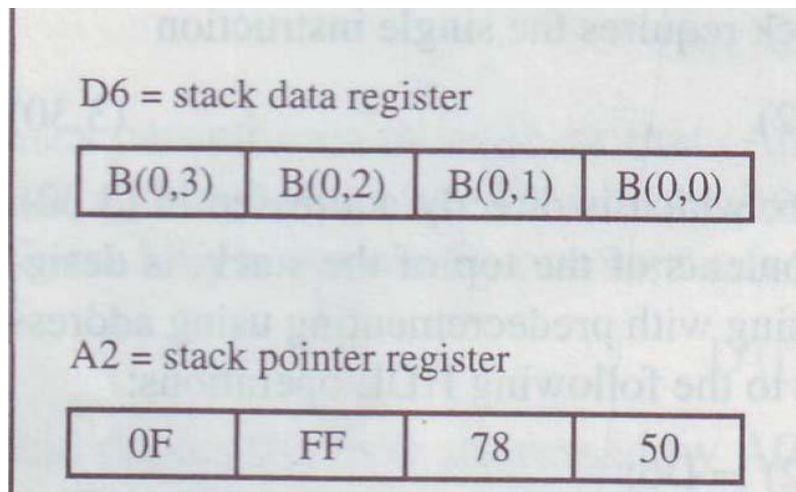
After the execution of MOVE.L D6, -(A2)

Number of Addresses

- **Three-address instruction:**
  ADD Z, X, Y
  Z:= X+Y.

- **Two-address instruction:**
  ADD X, Y
  X:= X+Y.

- **One-address instruction:**
  ADD X
  AC:=AC+X.

Number of Addresses

Implement X:=A×B+C×C, where A, B, C and X are stored in the memory.

Instruction	Comment
LOAD A	AC:=A
MULTIPY B	AC:= AC * B
STORE T	M(T):= AC
LOAD C	AC:= C
MULTIPLY C	AC:=AC * C
ADD T	AC:= AC + T
STORE X	M(X) : = AC

One address machine

Instruction	Comment
MOVE T, A	T:=A
MULTIPLY T, B	T:= T * B
MOVE X, C	X:=C
MULTIPLY X, C	X:= X * C
ADD X, T	X:=X+T

Two address machine

Instruction	Comment
MULTIPLY T, A, B	T:= A * B
MULTIPLY X, C, C	X:= C * C
ADD X, X, T	X:= X + T

Three address machine

Accumulator Architectures

- **Instruction set:**

  add A, sub A, mult A, div A, . .

  . load A, store A

- **Example: A*B - (A+B*C)**

  load B

  mul C

  add A

  store D

  load A

  mul B

  sub D

B	B*C	A+B*C	A+B*C	A	A*B	A*B-(A+B*C)

Accumulator Architecture : Pros and Cons

- **Pros**
  - Very low hardware requirements
  - Easy to design and understand

- **Cons**
  - Accumulator becomes the bottleneck
  - Little ability for parallelism or pipelining
  - High memory traffic

Zero Address Machine (Stack Based Architecture)

- Addresses are eliminated by storing operands in a push-down stack.

- All the operands are required to be in the top locations in the stack.

- Stack pointer automatically keeps track of the stack top.

- Push and pop operations are needed to transfer data to and from the stack.

- For example, X+Y is invoked by ADD that causes the top two operands, which should be X and Y, to be removed from the stack and added. The resulting sum X+Y is then placed at the top of the stack.

Stack Architectures
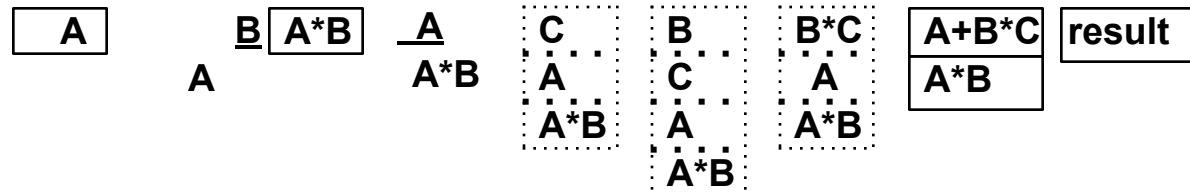
Example: A*B - (A+C*B)

push A
push B
mul
push A
push C
push B
mul
add
sub

| A |

| B | A*B |
| A |

| A |
| A*B |

| C |
| A |
| A*B |

| B |
| C |
| A |
| A*B |

| B*C |
| A |
| A*B |

| A+B*C |
| A*B |

| result |

Stack Architecture: Pros and Cons

- **Pros**
  - implicit operand addressing. Maintained by top of the stack.
  - Low hardware requirements.

- **Cons**
  - Stack becomes the bottleneck.
  - Little ability for parallelism or pipelining.
  - Data is not always at the top of stack when need, so additional instructions like POP and SWAP are needed.

The Requirements for Instruction Set

- It should be **complete**. We should be able to construct a machine language program to evaluate any function that is computable using a reasonable amount of memory space.

- It should be **efficient**. The frequently required functions can be performed rapidly using relatively few instructions.

- It should be **regular**. The instruction set should contain expected opcodes and addressing modes.

- The instruction should be **compatible** with those of

Classification of Instruction

- **Data transfer instructions** – Copy information. Example: *load, store, move* instruction.

- **Arithmetic instructions** – Perform operations on numerical data. Example: *add, sub, mul*, **div** etc.

- **Logical instructions** - Boolean and non-numerical operations. Example: **AND, NOT** etc.

- **Program control instructions** – It change the sequence in which program are executed. Example: Branch instruction.

- **Input-Output (IO) instructions** – It cause information to be transferred to or from external IO devices. Example: PRINT LINE.