Module 2:

Addressing modes and instruction set

Addressing modes:

It refers to the way in which the operand of an instruction is specified. [MSP430 has four basic modes for source but only 2 for destination in instruction with 2 operand.]

i) Double operand (format I)

ii) Single operand (format II)

iii) Jump → adds offset to pc

iv) RETI → format II without operand.

1) I) Register Addressing mode:

In this addressing mode both source and destination will be register in CPU.

eg: MOV.W R_5, R_6 (src, dst) : Copy the words from R_5 to R_6 register

It is the fastest mode and this instruction takes only one cycle

Any of the 16 registers (R_0 to R_{15}) can be used for either source or destination with some exceptions mentioned below.

i) The pc is incremented by 2 while the instruction is being fetched before it is used as a source.

ii) The Constant generator CG2 reads Zero as a source.

iii) Both the pc and sp address only words. So they must be even (lsb = 0). Therefore if pc and sp are used as destination lsb of src is discarded.

iv) The status register R_2 can be used either as either as source or destination

For byte instruction:

* The operands are taken from the lower byte. the upper byte is not affected
* The result is written to the lower byte of the registes and upper byte is cleared.

Note:

The upper byte of the registes can't be used as a source if needed the swpb instruction must be used

2) Indexed mode:

In this addressing mode the address is formed by adding (constant) base address to the contents of CPU register (index). The value in the registes is not changed.

This addressing mode can be used for both source, destination or both.

Eg: mov.b 3(R5), R6; load the byte from memory location 3+(R5) into R6
(3 = base address, R5 = index)

Bytes have no address restriction where as words must lie on even addresses only. CPU always calculate the index address in bytes.

3) Symbolic mode (pc - relative):

In this addressing mode the program counter pc is used as the base address and the constant is the offset to the data from pc. It is used by writing a symbol for memory location without any prefix.

Eg: i) mov.w label, R6; load / copy the word label into R6

The assembler replaces this instruction by the indexed form mov.w X(PC), R6; where X is a constant offset from the pc.

iii) mov.w l_1, l_2

⇓

mov.w X(PC), X(PC)

$X = l_1 - PC$

$Y = l_2 - PC$

where X is a offset where needs to be added to PC to get the label

This mode can be used both for source and destination.

The PC is automatically incremented to point to the next instruction

The PC relative addressing is required to produce position independent code.

Symbolic addressing is useful in application such as boot loaders and not appropriate for addresses that are fixed in memory map such as peripheral registers.

iv) Absolute addressing mode:

The constant in this form of indexed addressing is the absolute address of the data which is the complete address to be added to a register that contains Zero. This mode can be used for both source and destination.

Absolute addressing mode is shown identified by '&' symbol as prefix.

It should be used for special function and peripherals registers whose addresses are fixed in memory map.

PC is automatically incremented to point to the next instruction.

mov.b &P1IN, R6; load/copy the byte P1IN into R6.

The assembler replaces this instruction by indexed form

mov.b P1IN(SR) → offset ⇓ 0, R6;

where P1IN is absolute address of register

iii) SP - Relative (Indexed)

The SP can be used as register in indexed mode WKT the stack grows down in memory and SP points to the most recently added word (top of the stack)

Eg: MOV.W 2(SP), R6 ; Copy the most recent but one word from stack into R6 (pop)

2+SP → R6

This mode is only available for Source.

3) Indirect register mode:

This mode is available only for the Source and is identified by the Symbol at the @ as a prefix

Eg: @R6 → The Contents of R6 is used as address of the operand i.e R6 acts as a pointer.

Eg: MOV.W @R5, R6 ; Copy the word from the memory location / address pointed by R5 into R6

To implement indirect addressing for the destination we use the indexed form.

MOV.W R5, 0(R6) ; Copy the word from R5 into the memory location pointed by R6 (0+R6) = R6

0+R6 → address

A word of Zero must be Stored in the program memory and fetched from it.

4) Indirect autoincrement register mode

This mode is available only for the Source and is identified by Symbol '@' as a prefix and a '+' sign after the name of register

Eg: @R5+

mov.w @R5+, R6 ; Copy the word pointed by R5 into R6 and auto increment R5 by 2.

The value in R5 is used as pointer and automatically increments by 1 if a byte is fetched & automatically increments by 2 if a word is fetched.

This mode can't be used for destination instead the main instruction must use indexed mode with offset of zero followed by an increment instruction of one or 2 bytes as shown

```
mov.w   R6, 0(R5)
incd.w  R5
```

In all addressing modes the operations on the first address are fully completed before the second address is evaluated.

Eg: mov.w @R5+, 4(R5)
0X0100(R5)

Let R5 be 0006H, the contents of memory location 0006H is fetched and R5 is auto incremented by 2 i.e R5 = 0008H, this value of R5 is used in the destination address calculation i.e 4+8 = C.
The number fetched from source goes into this memory location.

5) Immediate addressing mode

This addressing mode is valid only for the source operand. The source operand is a constant. when fetching the source, the program counter points to the word following the instruction and moves the contents to the destination.

Eg: mov.w #0X1234, R5 ; Copy the immediate word from into register R5.

The assembler equivalent instruction is

mov.w @pc , R_5 ;

The pc is automatically incremented as soon as instruction is fetched.

Instruction set :

MSP430 has 27 native / core instructions. and 24 emulated instructions.

The instruction set is orthogonal. i.e all addressing modes can be used with all instructions and registers with a few exceptions.

.w indicates word operation
.b indicates byte operation
default is word operation.

The instruction set of MSP430 can be classified into 4 categories namely

1> Data transfer instructions
2> Arithmetic and logical instructions
3> Shift and rotate instructions
4> Flow of control instruction / Jump instructions

1> Data transfer instructions

There are 2 classifications

a> Movement instruction
b> Stack operation.

a> Movement instructions :

The mov instructions can address all of memory as either source or destination including both registers in the CPU and the whole memory map.

The general format of mov instruction is

mov.w (.b) src, dst ; Copy the word from source to the destination.

None of the flags in the Status register are affected by the mov instruction.

b) Stack operations: push and pop

The 2 Stack related instructions are push.w src; and pop.w dst;

push.w src ; push data onto stack → decrement sp and store the data
pop.w dst : pop data from stack.
[emulated] ↓
Take data and increment sp

*--sp

*sp++

The pop operation is emulated using post increment addressing but push requires a special instruction. because pre-decrement addressing is not available.

2) Arithmetic and logical instructions,

a) i) Arithmetic instructions with 2 operands:

The arithmetic flags namely Carry, Zero, -ve and overflow are affected by these instructions.

The Carry is treated as 'not borrow'

The arithmetic op instructions with 2 operations are listed below.

add.w src, dst ;		src + dst → dst
add with carry addc.w src, dst ;		src + dst + c → dst
emulated add carry addc.w dst ;		dst + c → dst
sub.w src, dst ;		dst − src → dst
Subtract with carry subc.w src, dst ;		dst − (src + nc) → dst
emulated Subtract carry sbc.w dst;		dst − nc → dst
cmp.w src, dst ;		dst − src

The Compare instruction is same as subtractions except that only the bits in the Status register are affected. the result is not written back to the destination.

ii b) Arithmetic instructions with one operand:

clr.w dst ;	dst = 0	
inc.w dst ;	dst = dst + 1	→ add.w #1, dst
incd.w dst ;	dst = dst + 2	→ add.w #2, dst
dec.w dst ;	dst = dst - 1	→ sub.w #1, dst
decd.w dst ;	dst = dst - 2	
tst.w dst ; (Compare with 0)	dst - 0.	

All these instructions are emulated which indicates the operand is always a destination.

The test & Compare instruction Compares destination with Zero and affects only the flags.

ii c) Decimal arithmetic:

These instructions are used only when the operands are binary coded decimal (BCD). Therefore each nibble range is 0 to 9.

dadd.w src, dst ; (decimal add with carry) src + dst + C → dst

[Emulated] dadc.w dst ; (decimal add carry) dst + C → dst

b) i) Logical operands instructions with 2 operands:

(bitwise and) and.w	src, dst ;	dst = src & dst.
(bitwise XOR) xor.w	src, dst ;	dst = src ∧ dst
(bitwise test) bit.w	src, dst ;	dst & src
(affects only flag)		
(bit set) bis.w	src, dst ;	dst = src | dst
(bit clear) bic.w	src, dst ;	dst = dst & ~src

Let R_5 = 5A9CH and R_6 = 112AH. write the results of all the operations. (Aug 21)

i) 5A9CH & 112AH

```
0101 1010 1001 1101
0001 0001 0010 1010
---------------------
0001 0000 0000 1000  → 1008H
```

Xor → 0101 1010 1001 1100
0001 0001 0010 1010

0100 1011 1011 0110

4BB6H

bis.w → Src | dst

0101 1010 1001 1100
0001 0001 0010 1010

0101 1011 1011 1110

5BBEH

bic.w → ~~0101~~ ~Src & dst

1010 0101 0110 0010
0001 0001 0010 1010

0000 0001 0010 0010

0122H

bit.w → dst & src

bit → Z=0, C=1 Zero flag is reset i.e Zero since result is not Zero.

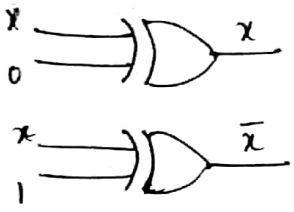
And and the bit test are identical except that the bit is only a test and doesn't change its destination. only flags are affected. The Z bit of flag is set as usual and the Carry flag C = ~ Z.

The bit set (bis) and the bit clear (bic) instructions are used with masks to set and clear bits. The bis and bic do not affect status bits bit operations are called read-modify-write operations because CPU can't operate on bits individually, it must read registers into ALU perform the operation and write the result back.

ii) Logical instruction with one operand

Invert bit inv.w dst ; dst = ~dst [emulated]

This instruction is emulated using XOR and inherits its property C = ~Z



xor.w Src, dst
xor.w #ffff, dst

Eg: Let R_5 = 5A5A = dst

~R_5 = A5A5

5A5A = 0101 1010 0101 1010
1111 1111 1111 1111
1010 0101 1010 0101

~5A5A = A5A5H

Byte manipulation

These instruction doesn't have suffix (.w & .b) since size of the operand is fixed.

Swap bytes 1) swpb src ; Swap upper and lower bytes of src

Sign extend 2) sxt src ; Extend sign of lower byte

The sxt is used to convert signed byte to signed word. It copies the value of bit 7 (sign of lower byte) into bits 8 to 15 and makes Carry=~Z

The truncation from a word to a byte can be done using mov.b instruction.

Eg: $R_5 = Src = AB95H$

$R_5 = Src = 95H$

1111 1111 1001 0100

FF95H

operations on bits in Status register

There is a set of emulated instructions to set or clear the lowest four bits in the status register.

emulated

- clrc → clear Carry bit
- clrn → clear negative bit
- clrz → clear Zero bit
- Setc → set Carry bit
- Setn → Set negative bit
- Setz → Set Zero bit
- dint → disable gie
- eint → enable gie

C = 0 → bic #1, R_2

N = 0 → bic #2, R_2

Z = 0 → bic #4, R_2

C = 1 → bis #1, R_2

N = 1 → bis #2, R_2

Z = 1 → bis #4, R_2

GIE = 0 → bic #8, R_2

GIE = 1 → bis #8, R_2

3) Shift and rotate instructions

There are 3 types of shifts namely

i) Logical shift → Inserts Zeros both right and left shift

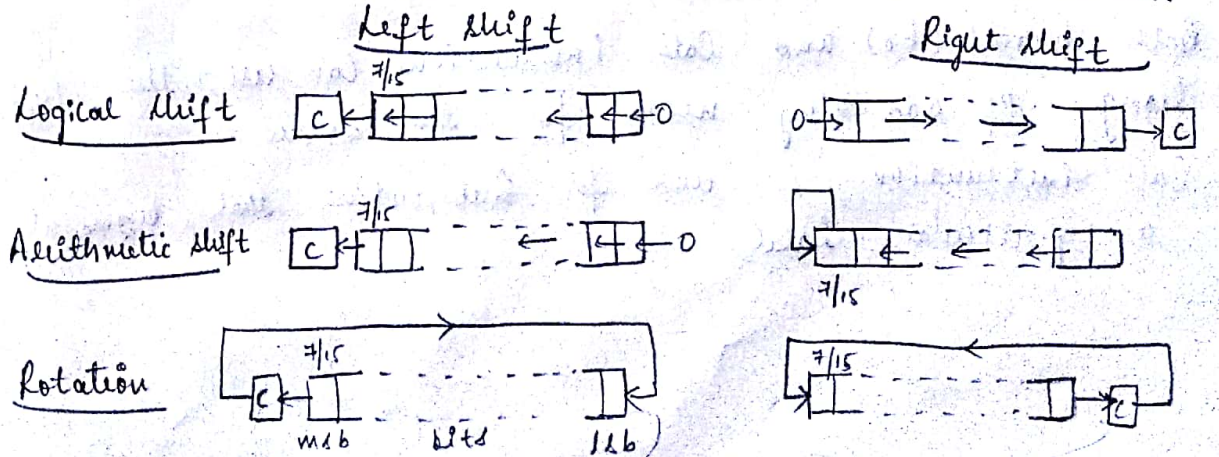
ii) Arithmetic shift → Inserts Zeros for left shifts but the msb (signed bit) is replicated for right shift.

iii) Rotation shift

↓

Doesn't loose any bits only their position changes.

The pictorial representation of the shifts is as shown



MSP430 has arithmetic shifts and rotations, all of which use the Carry bit.

The right shifts are native instructions but the left shifts are emulated instructions.

The 4 shift and rotate instructions are

[emulated] rla dst ; arithmetic left shift
rra src ; arithmetic right shift
[emulated] rlc dst ; rotate left through carry
rrc src ; rotate right through carry

Logical left shift is similar to the arithmatic left shift. A logical right shift can be emulated by first clearing Carry bit and then rotating right using 'rrc' instruction

4> Flow of Control:

The Subroutine interrupts and branch instructions alter the flow of control of an instruction.

The instructions related to branches, subroutines and interrupts are listed below.

[emulated] br src; branch to the specified label (mov.w #label, pc)
Call src; Call Subroutine
[emulated] ret ; return from Subroutine
[emulated] nop ; no operation (single cycle instruction
reti ; return from interrupt

Both branch (br) and Call instructions can use the full range of addressing modes for the source.

Call instruction is used for Subroutine that begins at a particular label.

Both The label is translated by the assembler to address of the first instruction in Subroutine. which is value to be loaded into the program counter to Call the Subroutine. i.e CALL #label

The branch instruction BR label is emulated, this instruction branch label is converted into (emulated) translated mov.w #label, pc

mov.w #label, pc.

Jump instructions

There are 2 kinds of jump

1) unconditional jump
2) Conditional jump

Unconditional jump

Jmp label; unconditional jump

The range of jump is ± 1KB from the current location. (It is a single word including offset)

branch can go anywhere in the address space of memory space and can use any addressing mode but is slower and requires an extra word of program storage.

$ indicates the current value of pc.

jump $ indicates an infinite loop.

Conditional jumps or decision making instructions

These instructions test a condition if the condition is satisfied, the jump to the specified label is performed.

The various conditional jumps in MSP 430 are

Used in Testing flags

jc	label;	jump if Carry Set	C=1
jnc	label;	jump if Carry not Set	C=0
jz	label;	jump if Zero	Z=1
jnz	label;	jump if non-Zero	Z=0
jn	label;	jump if negative	N=1

Conditional jumps after instruction that use C, Z, N flags

Conditional jumps to be used after Compare instructions

Similar to

(jz)	jeq	label;	jump if equal	dst == src
(jnz)	jne	label;	jump if not equal	dst != src
(jc)	jhs	label;	jump if higher or Same	dst ≥ src
(jnc)	jlo	label;	jump if lower	dst < src

To Compare

Conditional jumps to test the overflow bit(V) for Signed values

jge	label;	jump if greater or equal	dst >= src
jl(t)	label;	jump if less than	dst < src