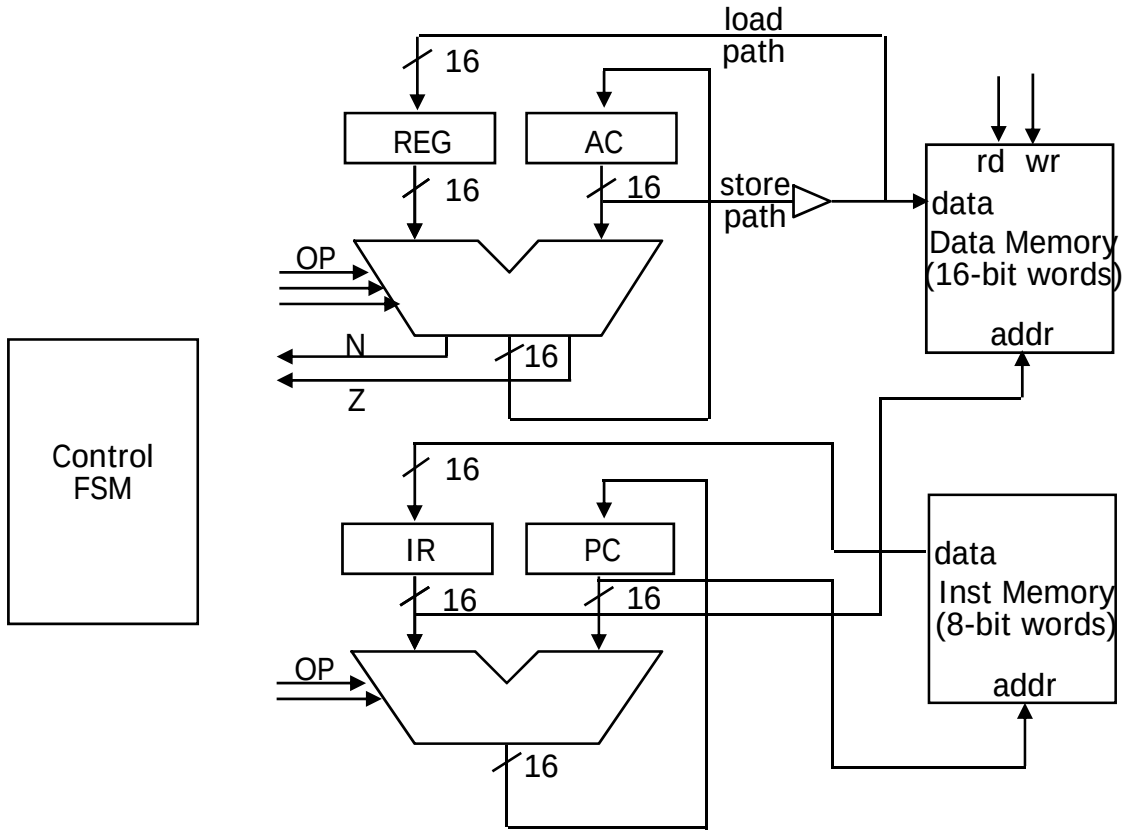
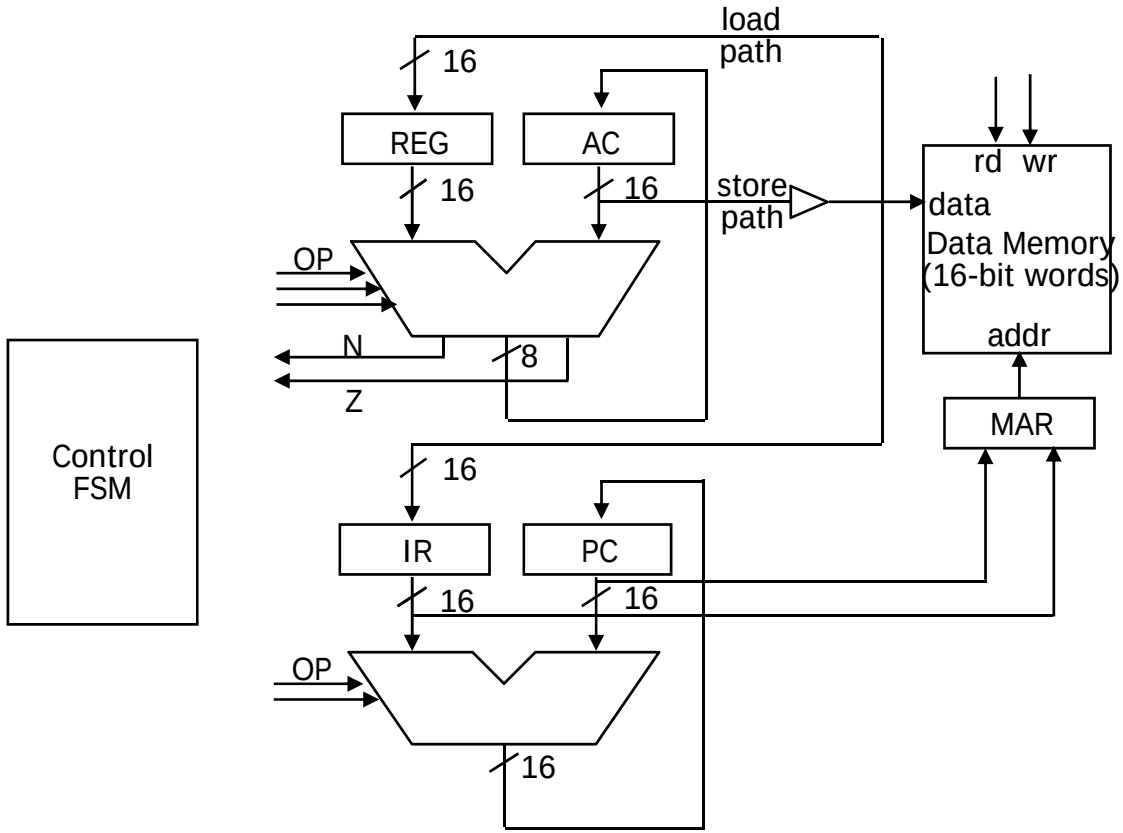
Block diagram of processor (Harvard)

- Register transfer view of Harvard architecture
  - Separate busses for instruction memory and data memory
  - Example: PIC



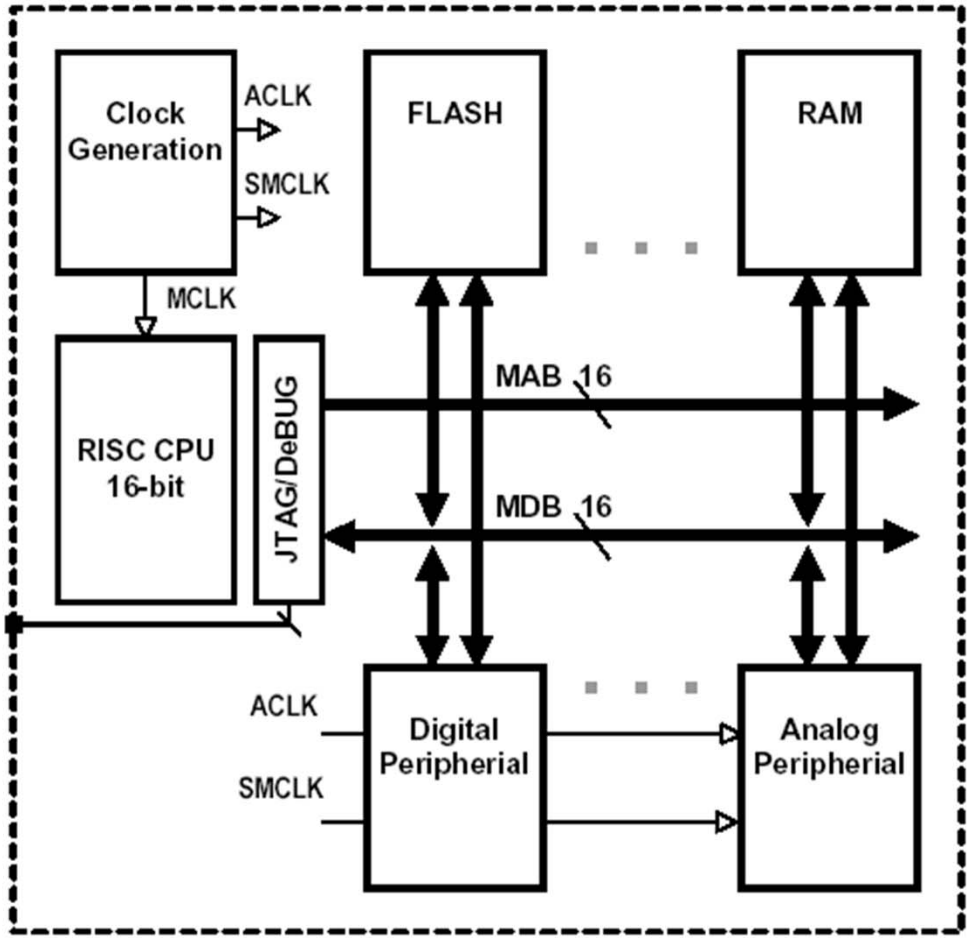
Block diagram of processor (Princeton)

- Register transfer view of Princeton / von Neumann architecture
  - Single unified bus for instructions, data, and I/O
  - Example: MSP430



MSP 430 Modular Architecture

von-Neumann common bus connects CPU to all memory and peripherals



Architecture reduces power consuming, noise generating fetches to memory

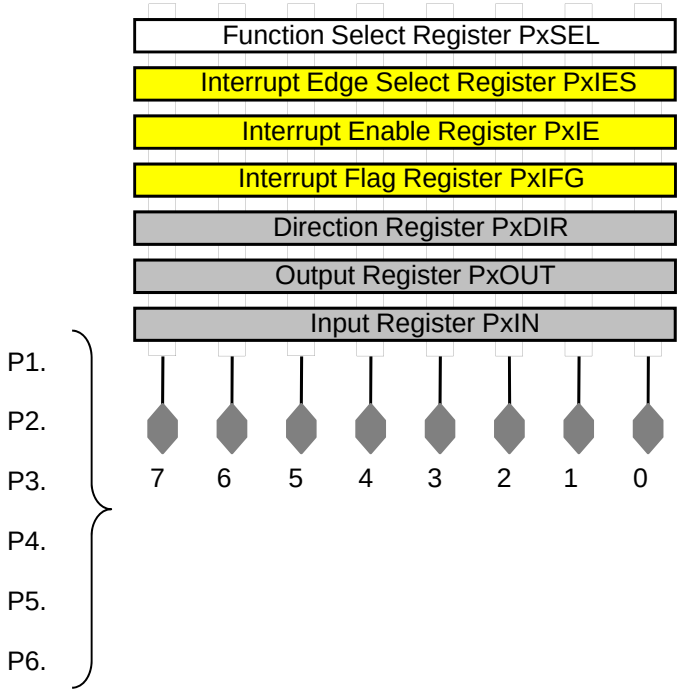
Embedded emulation accessed in-application with JTAG

16-bit bus handles wide-width data much more effectively

Digital I/O

Independently programmable individual I/Os

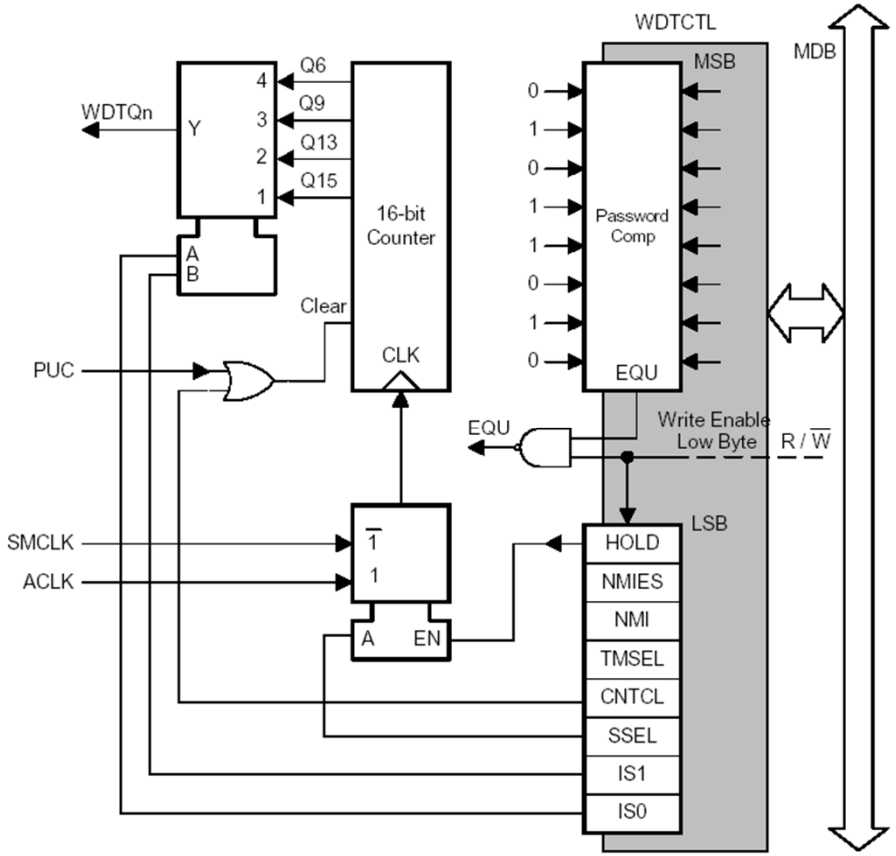
- Up to 6 ports (P1 – P6)
- Each has 8 I/O pins
- Each pin can be configured as input or output
- P1 and P2 pins can be configured to assert an interrupt request



Port1 Port2	Port3 ... Port6
yes	yes
yes	no
yes	no
yes	no
yes	yes
yes	yes
yes	yes

Watchdog Timer

WDT module performs a controlled system restart after a software problem occurs



- Can serve as an interval timer (generates interrupts)
- WDT Control register is password protected
- Note: Powers-up active

Example 1: msp430x20x3_1.c

```
//  MSP430F20xx Demo - Software Toggle P1.0
#include  <msp430x20x3.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
  P1DIR |= 0x01;   // Set P1.0 to output direction
for (;;)
  {
    volatile unsigned int i;
    P1OUT ^= 0x01; // Toggle P1.0 using XOR
    i = 50000;  // Delay
    do (i--);
    while (i != 0);
  }
}
```

WDTCTL:
Watch Dog Timer ConTroL

WDTPW:
WDT PassWord

WTDHOLD:
WDT Hold

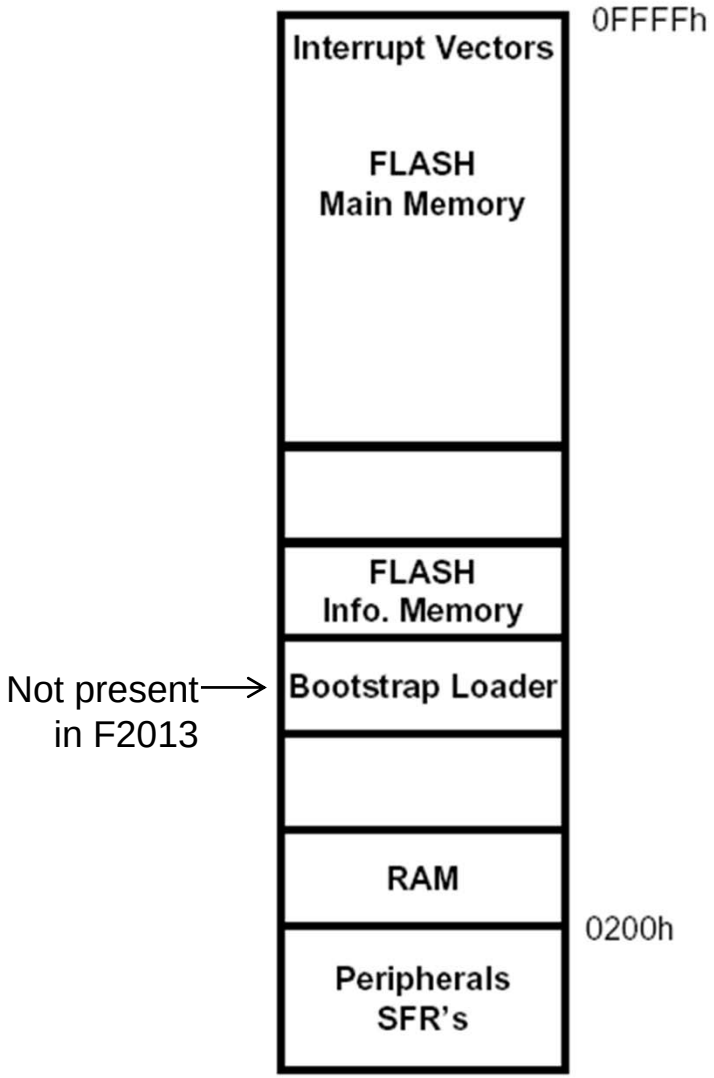
P1DIR:
Port 1 DIRection

P1OUT:
Port 1 OUTput state

Example 2: msp430x20x3P1_01.c

```
//  MSP430F20xx Demo - Software Poll P1.4, Set P1.0 if P1.4 = 1
//  Desc: Poll P1.4 in a loop, if hi P1.0 is set, if low, P1.0 reset.
//  ACLK = n/a, MCLK = SMCLK = default DCO
#include  <msp430x20x3.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
  P1DIR |= 0x01;            // Set P1.0 to output direction
  while (1)                           // Test P1.4
  {
    if ((0x10 & P1IN)) P1OUT |= 0x01;// if P1.4 set, set P1.0
    else P1OUT &= ~0x01;              // else reset
  }
}
```

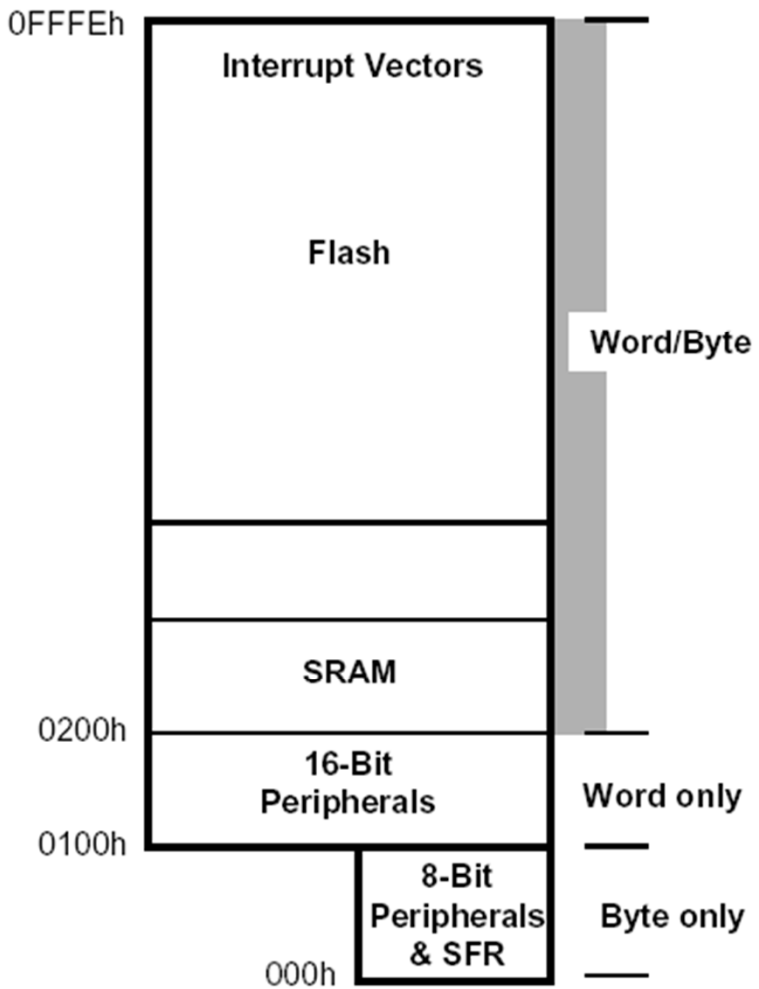
MSP430 Memory Model

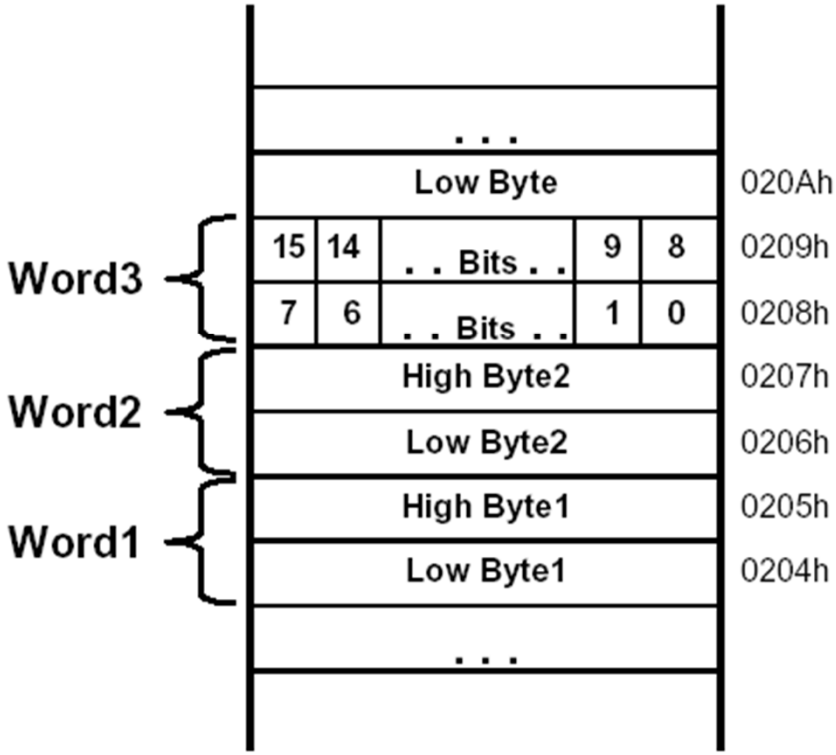


- Unified 64kB continuous memory map
- Same instructions for data and peripherals
- Program and data in Flash or RAM with no restrictions
- Easy to understand with no paging
- Designed for modern programming techniques such as pointers and fast look-up tables

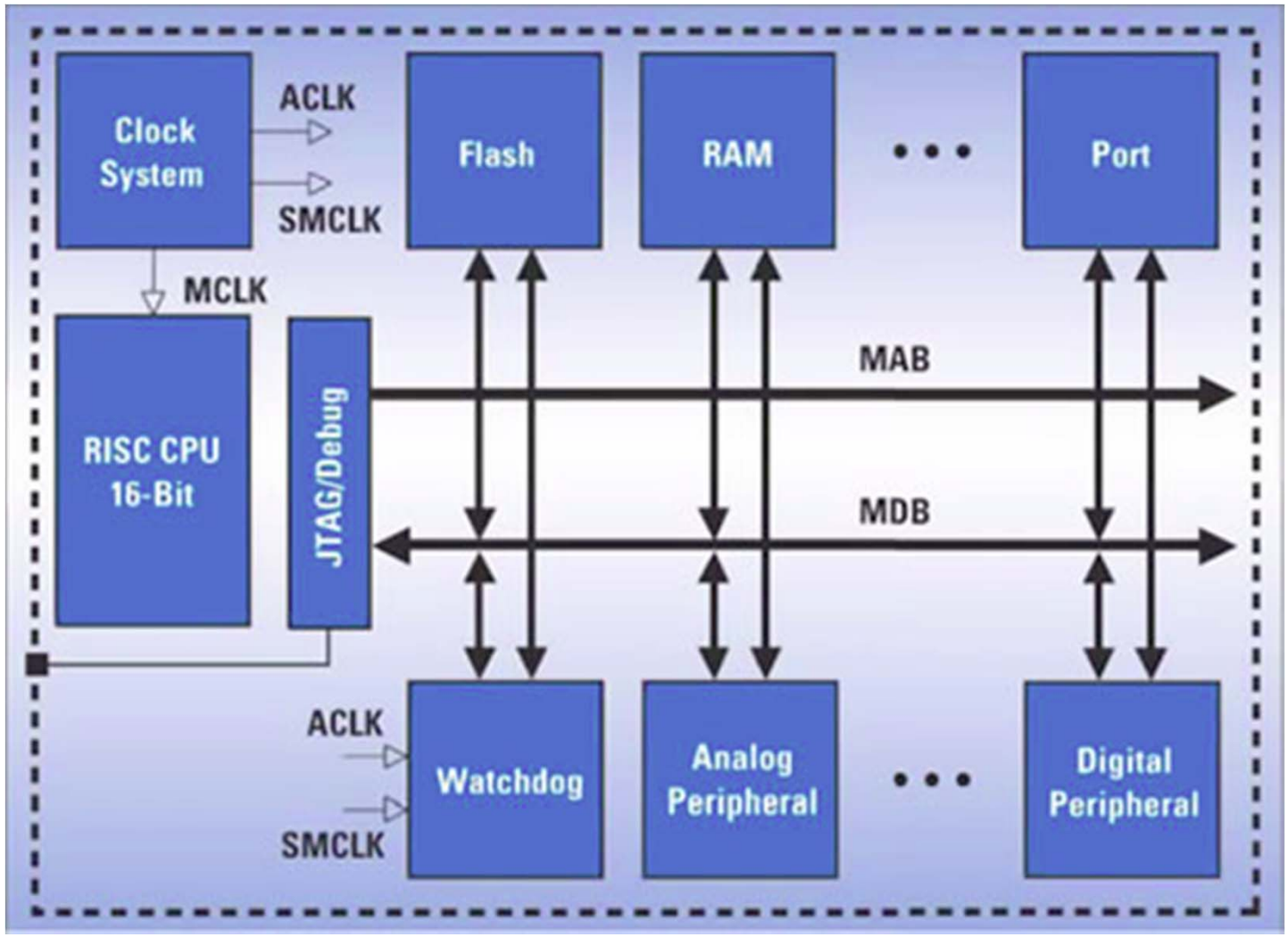
SFR: Special Function Registers

Memory Organization



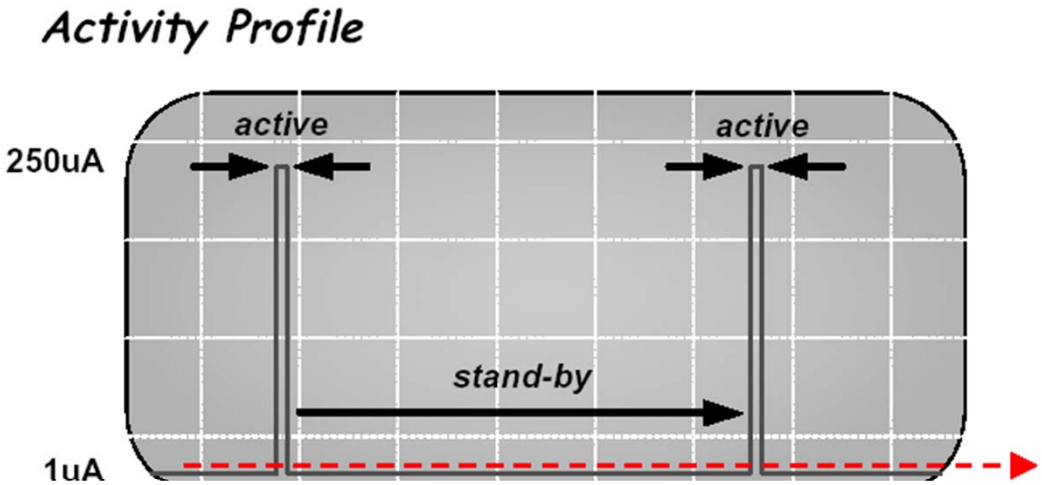


MSP 430 Architecture: A Closer Look

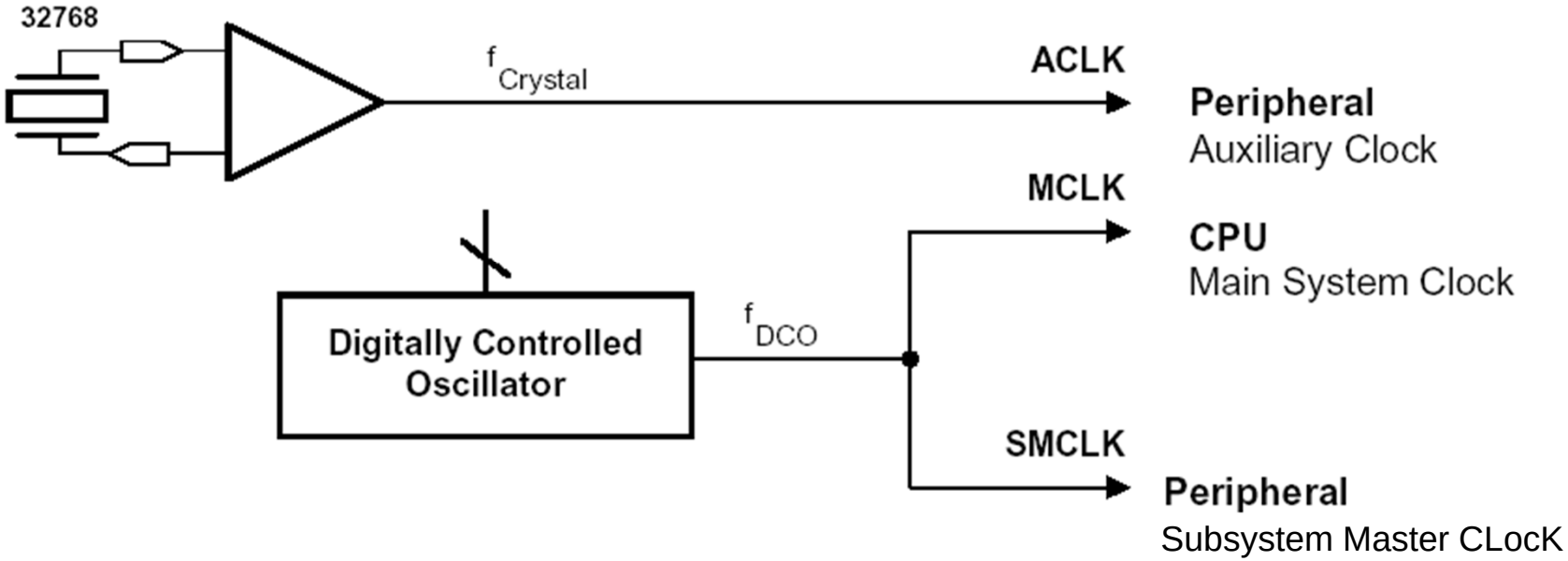


Basic Clock System

Basic Clock Module provides the clocks for the MSP430 devices



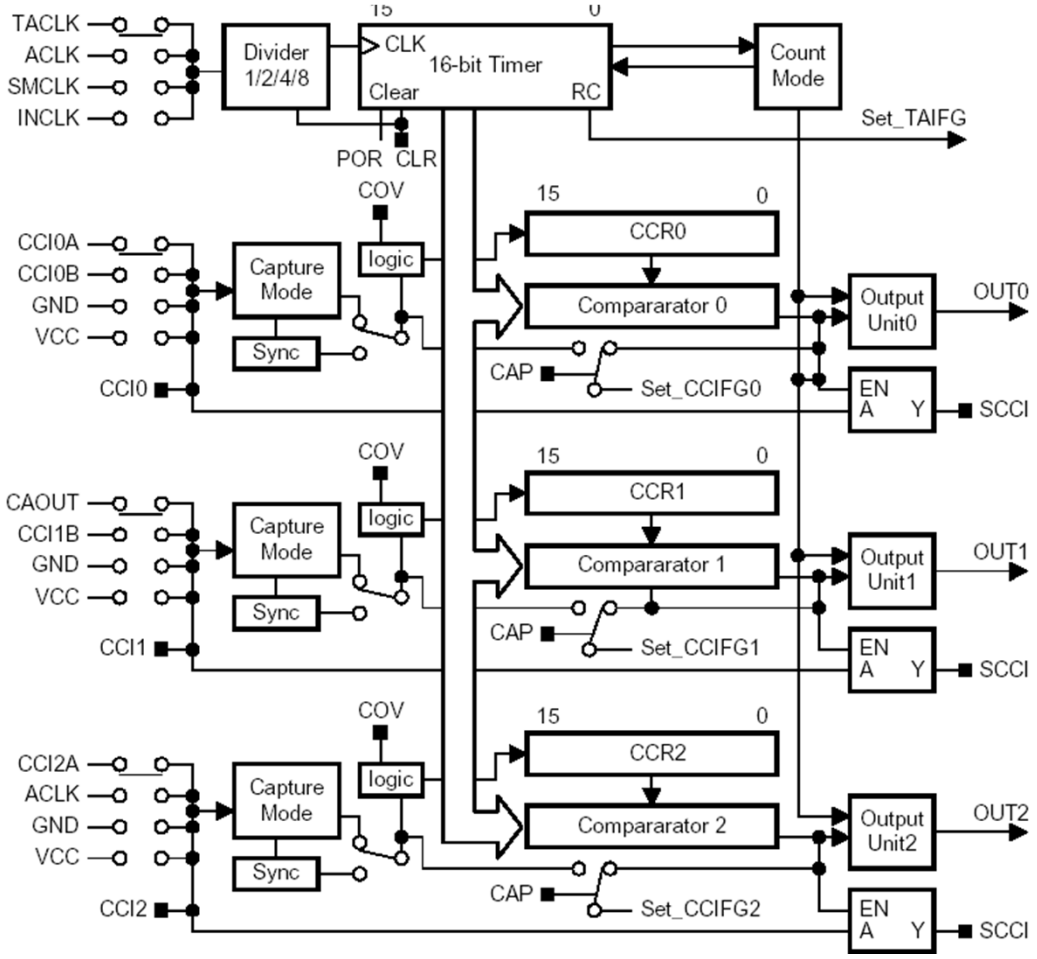
Or VLO (in MSP430F2xx)



Timer_A

Timer_A is a 16-bit timer/counter with multiple capture/compare registers (2 in F2013)

- Capture external signals
- Compare PWM mode



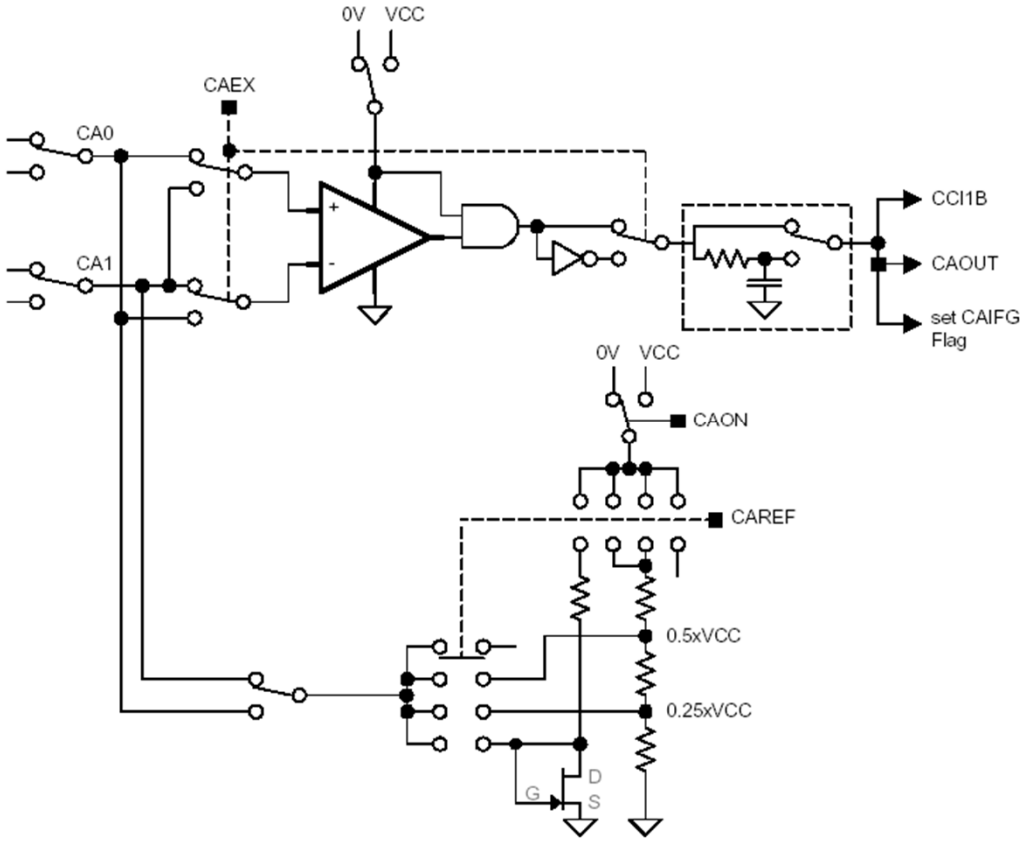
Example 3: msp430x20x3_ta_01.c

```
//  MSP430F20xx Demo - Timer_A, Toggle P1.0, CCR0 Cont. Mode ISR, DCO SMCLK
//  Description: Toggle P1.0 using software and TA_0 ISR. Toggles every
//  50000 SMCLK cycles. SMCLK provides clock source for TACLK.
//  During the TA_0 ISR, P1.0 is toggled and 50000 clock cycles are added to
//  CCR0. TA_0 ISR is triggered every 50000 cycles. CPU is normally off and
//  used only during TA_ISR.
//  ACLK = n/a, MCLK = SMCLK = TACLK = default DCO
#include <msp430x20x3.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  P1DIR |= 0x01;                            // P1.0 output
  CCTL0 = CCIE;                             // CCR0 interrupt enabled
  CCR0 = 50000;
  TACTL = TASSEL_2 + MC_2;                  // SMCLK, contmode
  _BIS_SR(LPM0_bits + GIE);                 // Enter LPM0 w/ interrupt
}
// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  P1OUT ^= 0x01;                            // Toggle P1.0
  CCR0 += 50000;                            // Add Offset to CCR0
}
```

Comparator_A

Comparator_A is an analog voltage comparator

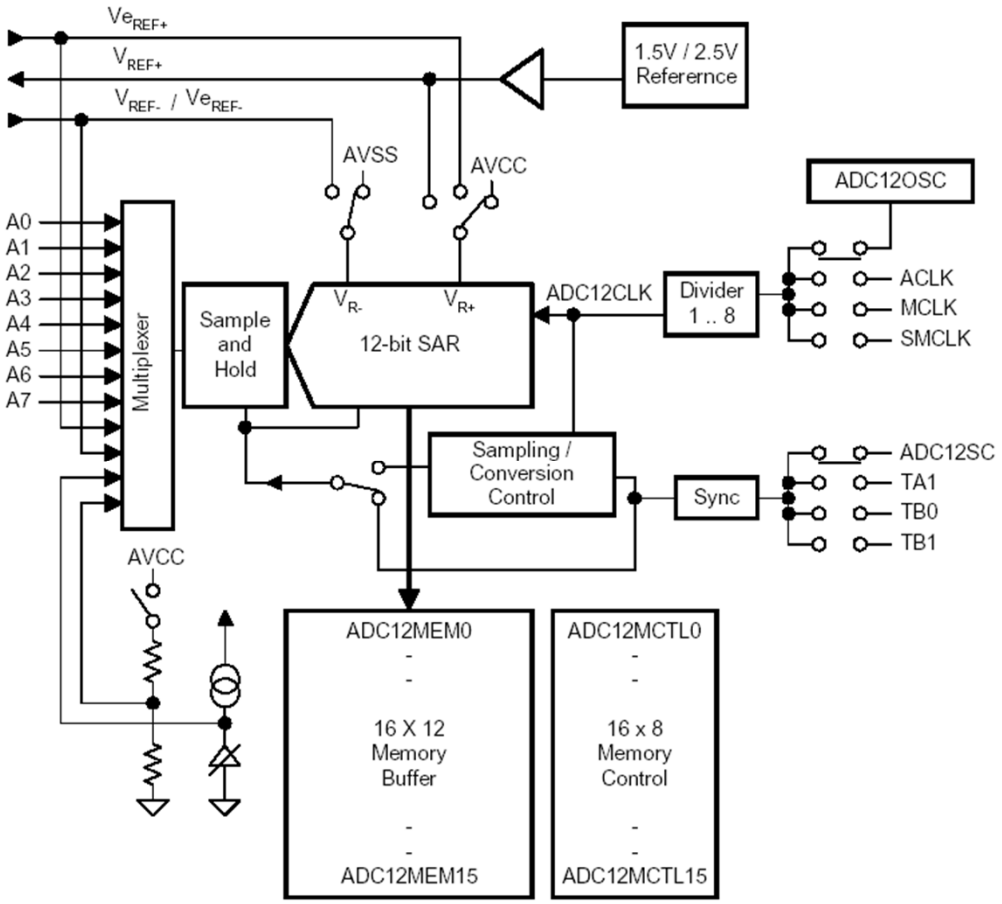
- Supports precision slope analog-to-digital conversions
- Supply voltage supervision, and
- Monitoring of external analog signals.



ADC12

High-performance 12-bit analog-to-digital converter

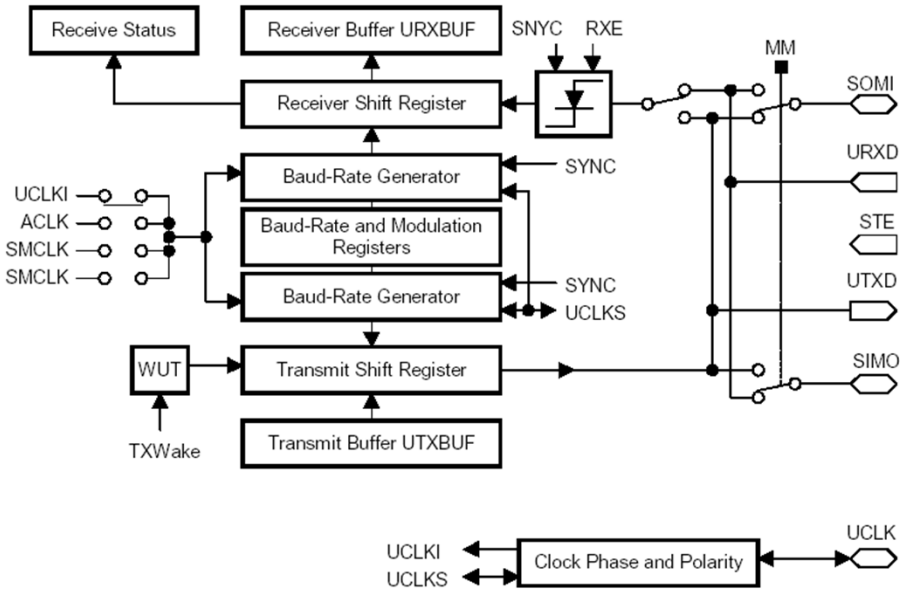
- More than 200 Ksamples/sec
- Programmable sample& hold
- 8 external input channels
- Internal storage



USART Serial Port

The universal synchronous/ asynchronous receive/transmit (USART) peripheral interface supports two serial modes with one hardware module

- UART or SPI (Synchronous Peripheral Interface) modes
- Double-buffered
- Baud-rate generator



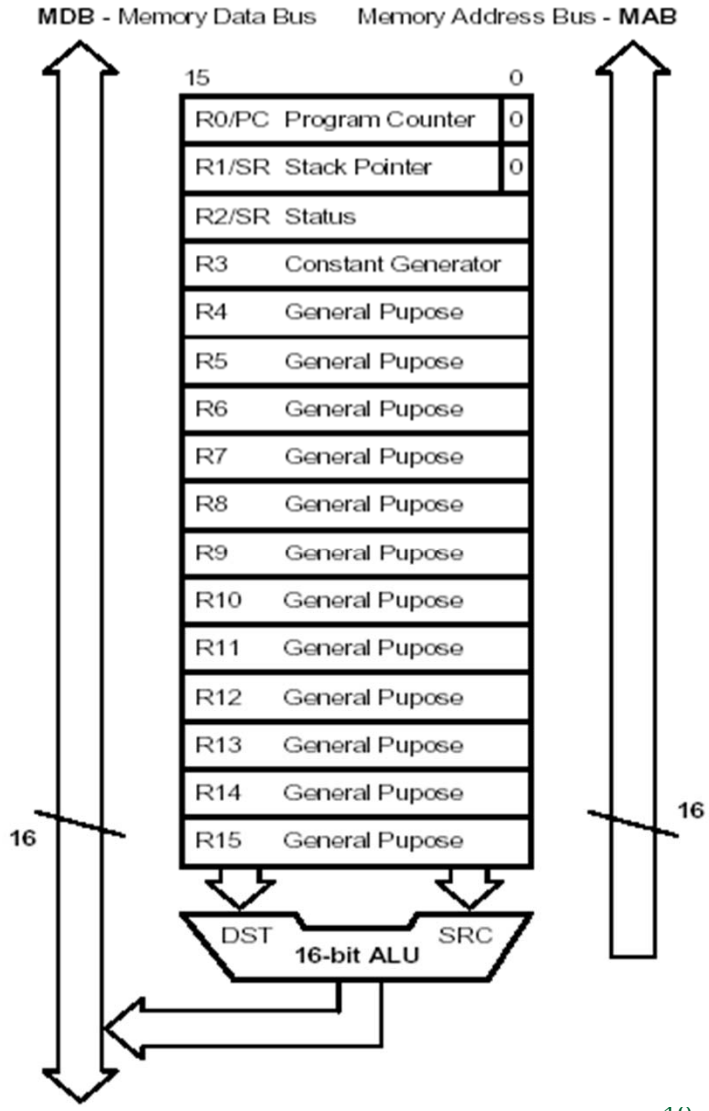
End of lecture 2

MSP430 CPU Introduction

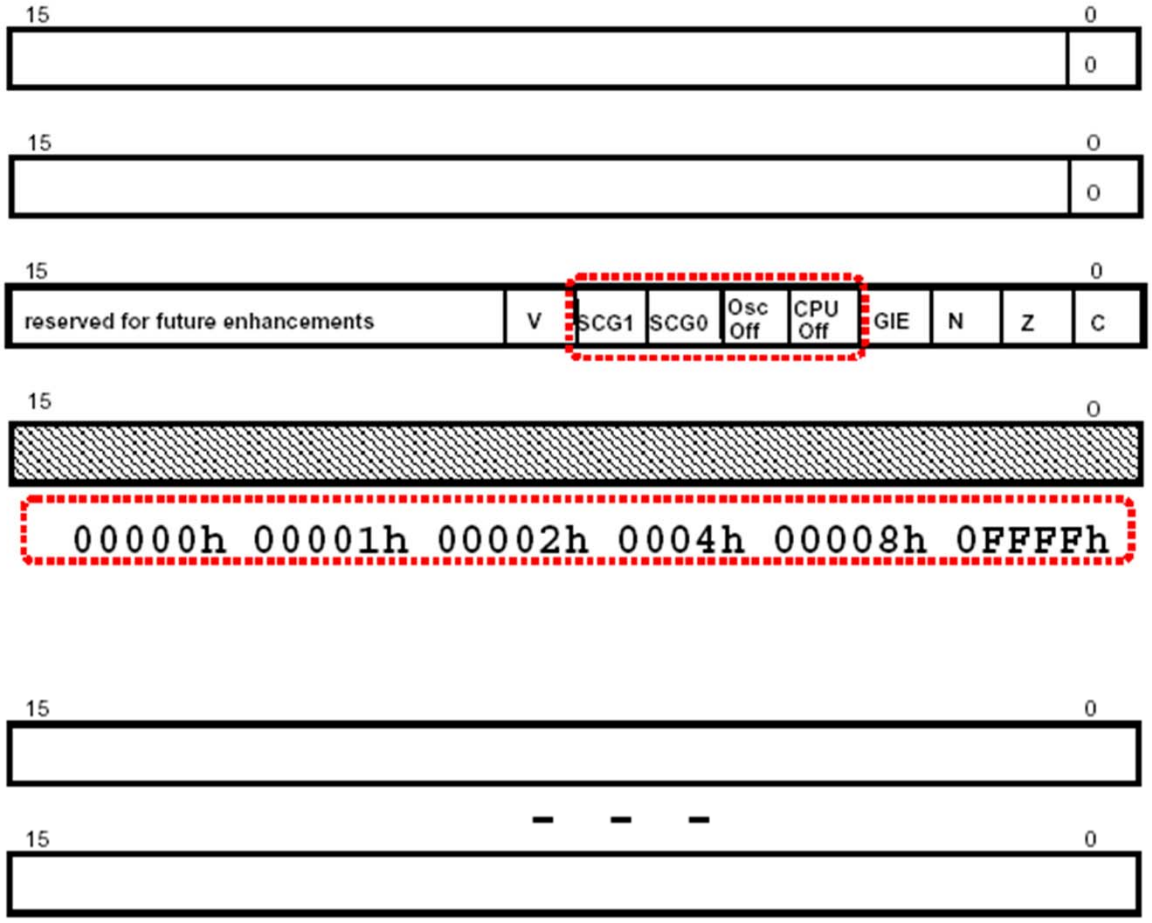
- RISC architecture with 27 instructions and 7 addressing modes.
- Orthogonal architecture with every instruction usable with every addressing mode.
- 16-bit address bus allows direct access and branching throughout entire memory range; no paging
- 16-bit data bus allows direct manipulation of word-wide arguments.
- Constant generator provides six most used immediate values and reduces code size.
- Direct memory-to-memory transfers without intermediate register holding.
- Word and byte addressing and instruction formats.
- Compact silicon 30% smaller than an ‘8051 saves power and cost

MSP430 register properties

- Large 16-bit register file eliminates single accumulator bottleneck, reduces fetches to memory.
- Full register access including program counter, status registers, and stack pointer.
- Single-cycle register operations.



MSP430 Registers



R0 - PC Program Counter
16-bit = no paging

R1 - SP Stack Pointer
Addressable = great "C" code

R2 - SR Status Register
Define LPMx

R3/R2 - CG Constant Generator
automatic generation of common used values reduces code size 30%

R4 - General Purpose

R15 - General Purpose

R4 through R15 are single-cycle, general purpose and identical in all respects - used for math, storage, and addressing modes.

Registers: PC (R0)

- Each instruction uses an even number of bytes (2, 4, or 6)
- PC is word aligned (the LSB is 0)

```
Format:
OP  SRC, DST ; NB, a lot of processors use "DST,SRC"

MOV #LABEL,PC ; Branch to address LABEL
MOV LABEL,PC ; Branch to address contained in LABEL
MOV @R14,PC ; Branch indirect, indirect R14
```

Registers: SP (R1)

- Stack pointer for return addresses of subroutines and interrupts
- Stack is in general purpose RAM (unlike e.g. PIC)
- SP is word aligned (the LSB is 0)
  - Pushing a single byte on the stack wastes a byte…the byte at the "wasted" location is not modified by the push
  - Contrast with single-byte register operations, which zero the unused byte
- SP points to most recently added word (not to next free word)
- Starts from top of memory (0x0280 in F2013)
- In assembly, you must initialize the SP yourself!
  - Don't try to jump in to an initialization subroutine to set up the SP…if the SP is not set up before the subroutine call, you won't return properly from the initialization subroutine!

Registers: SP (R1)

Example stack manipulations:

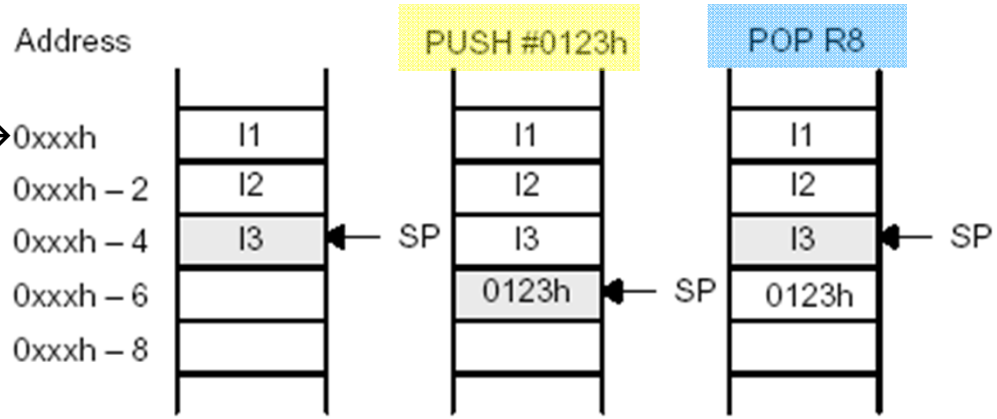
```
MOV 2(SP),R6 ; Item I2 -> R6
MOV R7,0(SP) ; Overwrite TOS with R7
PUSH #0123h ; Put 0123h onto TOS
POP R8 ; R8 = 0123h
```

2(SP) means 2+SP

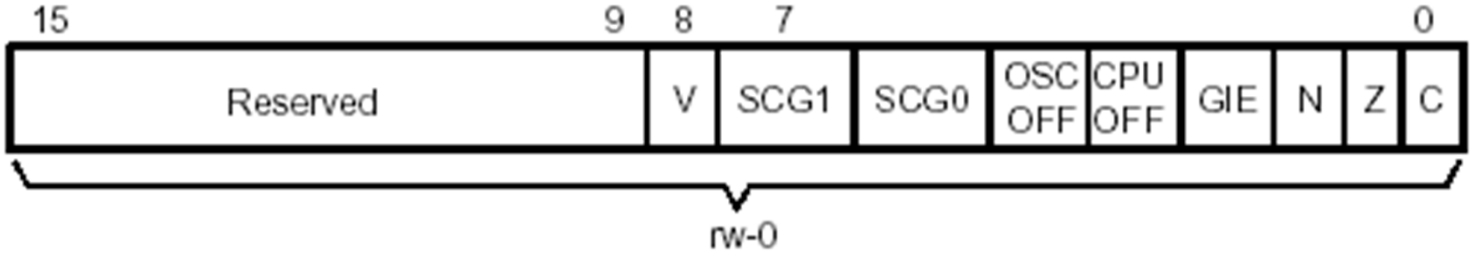
Highest RAM address is 0x0280 in F2013 →

TOS: "Top Of Stack"

*TOS is actually the **lowest** address in the stack!*



Registers: SR (R2)



- C: SR(0) Carry
- Z: SR(1) Zero
- N: SR(2) Negative (==msb of result)
- GIE: SR(3) Global interrupt enable
- CPUOff: SR(4) LPM control
- OSCOff: SR(5) LPM control
- SCG0: SR(6) System Clock Generator 0…LPM control
- SCG1: SR(7) System Clock Generator 0…LPM control
- V: SR(8) signed oVerflow

Example: msp430x20x3_1.asm

```
.cdecls C,LIST,  "msp430x20x3.h"

;-------------------------------------------------------------------------------
            .text                           ; Progam Start
;-------------------------------------------------------------------------------
RESET       mov.w   #0280h,SP               ; Initialize stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
SetupP1     bis.b   #001h,&P1DIR            ; P1.0  output
                                            ;
Mainloop    xor.b   #001h,&P1OUT            ; Toggle P1.0
Wait        mov.w   #50000,R15              ; Delay to R15
L1          dec.w   R15                     ; Decrement R15
            jnz     L1                      ; Delay over?
            jmp     Mainloop                ; Again
                                            ;
;-------------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET                   ;
            .end
```

Status bits

Bit	Description	
V	Overflow bit. This bit is set when the result of an arithmetic operation overflows the signed-variable range.	
	`ADD(.B),ADDC(.B)`	Set when: Positive + Positive = Negative Negative + Negative = Positive, otherwise reset
	`SUB(.B),SUBC(.B),CMP(.B)`	Set when: Positive – Negative = Negative Negative – Positive = Positive, otherwise reset
SCG1	System clock generator 1. This bit, when set, turns off the SMCLK.	
SCG0	System clock generator 0. This bit, when set, turns off the DCO dc generator, if DCOCLK is not used for MCLK or SMCLK.	
OSCOFF	Oscillator Off. This bit, when set, turns off the LFXT1 crystal oscillator, when LFXT1CLK is not use for MCLK or SMCLK	
CPUOFF	CPU off. This bit, when set, turns off the CPU.	
GIE	General interrupt enable. This bit, when set, enables maskable interrupts. When reset, all maskable interrupts are disabled.	
N	Negative bit. This bit is set when the result of a byte or word operation is negative and cleared when the result is not negative.	
	Word operation:	N is set to the value of bit 15 of the result
	Byte operation:	N is set to the value of bit 7 of the result
Z	Zero bit. This bit is set when the result of a byte or word operation is 0 and cleared when the result is not 0.	
C	Carry bit. This bit is set when the result of a byte or word operation produced a carry and cleared when no carry occurred.	

NB: in MSP430, mov operations do not affect status register

Demo interlude, including debugger!

Constant Generators

- As – source register addressing mode in the instruction word

Register	As	Constant	Remarks
R2	00	– – – – –	Register mode
R2	01	(0)	Absolute address mode
R2	10	00004h	+4, bit processing
R2	11	00008h	+8, bit processing
R3	00	00000h	0, word processing
R3	01	00001h	+1
R3	10	00002h	+2, bit processing
R3	11	0FFFFh	–1, word processing

Constant generator examples

```
Address    Assembled instr    Viewed in debugger    ; Original user's asm code
0xF81C:    434F               CLR.B   R15          ; mov.b  #0,R15
0xF81E:    435F               MOV.B   #1,R15       ; mov.b  #1,R15
0xF820:    436F               MOV.B   #2,R15       ; mov.b  #2,R15
0xF822:    407F 0003          MOV.B   #0x0003,R15  ; mov.b  #3,R15
0xF826:    426F               MOV.B   #4,R15       ; mov.b  #4,R15
0xF828:    407F 0080          MOV.B   #0x0080,R15  ; mov.b  #80h,R15
0xF82C:    407F 007F          MOV.B   #0x007f,R15  ; mov.b  #7Fh,R15
0xF830:    430F               CLR.W   R15          ; mov.w  #0,R15
0xF832:    431F               MOV.W   #1,R15       ; mov.w  #1,R15
0xF834:    432F               MOV.W   #2,R15       ; mov.w  #2,R15
0xF836:    403F 0003          MOV.W   #0x0003,R15  ; mov.w  #3,R15
0xF83A:    422F               MOV.W   #4,R15       ; mov.w  #4,R15
0xF83C:    403F 0080          MOV.W   #0x0080,R15  ; mov.w   #80h,R15
0xF840:    433F               MOV.W   #-1,R15      ; mov.w  #0xFFFF,R15
```

Move operations that can use the constant generator assemble to 1 word

Move operations that cannot use the constant generator assemble to 2 words

CISC / RISC Instruction Set

- **Three instruction formats**
  - Source, destination
  - Destination
  - Jumping

Good summary of MSP430 instructions:
http://cnx.org/content/m23503/latest/

- **Fifty-one instructions available in assembler**
  - 27 basic instructions ⇨ *RISC*
  - 24 emulated instructions ⇨ *CISC*
- **Seven addressing modes for source, four for destination**

	source	destination
Register Mode	✓	✓
Indexed Mode	✓	✓
Symbolic Mode	✓	✓
Absolute Mode	✓	✓
Indirect Mode	✓	
Indirect-autoincrement Mode	✓	
Immediate Mode	✓	

- **Bit, byte and word processing**

27 Core RISC Instructions

Format I Source,Destination	Format II Single Operand	Format III +/- 9bit Offset
add(.b)	call	jmp
addc(.b)	swpb	jc
and(.b)	sxt	jnc
bic(.b)	push(.b)	jeq
bis(.b)	reti	jne
bit(.b)	rra(.b)	jge
cmp(.b)	rrc(.b)	jl
dadd(.b)		jn
mov(.b)		
sub(.b)		
subc(.b)		
xor(.b)		

Notes:
- .b or .w: byte or word
- add / addc: without/with carry
- dadd: decimal add w/ carry (→bin. coded decimal math)

swpb: swap bytes
sxt: sign extend (copies sign bit from b7 to b15-b8…for converting signed byte into signed word)

rra: Roll right
rrc: Roll right, through carry

Emulated Instructions

- Simply easier to understand with no code size or speed penalty
- Replaced by assembler with core instructions using CG, PC and SP

```
clrc                    ; Clear carry (emulated)
bic.w    #01h,SR        ; Core instruction

dec.w    R4             ; Decrement (emulated)
sub.w    #01,R4         ; Core instruction

ret                     ; Return (emulated)
mov.w    @SP+,PC        ; Core instruction
```

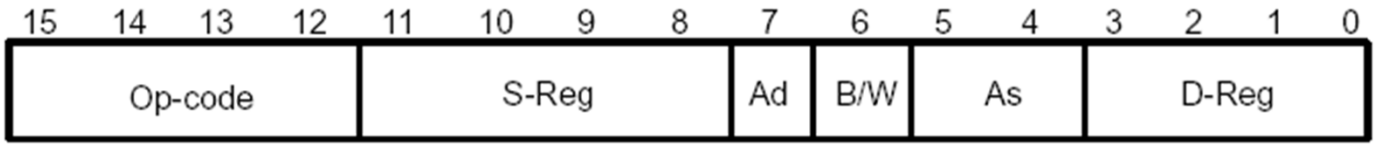
51 Total Instructions

Format I Source,Destination	**Format II** Single Operand	**Format III** +/- 9bit Offset	Support
add(.b)	**br**	jmp	**clrc**
addc(.b)	call	jc	**setc**
and(.b)	swpb	jnc	**clrz**
bic(.b)	sxt	jeq	**setz**
bis(.b)	push(.b)	jne	**clrn**
bit(.b)	**pop.(b)**	jge	**setn**
cmp(.b)	rra(.b)	jl	**dint**
dadd(.b)	rrc(.b)	jn	**eint**
mov(.b)	**inv(.b)**		**nop**
sub(.b)	**inc(.b)**		**ret**
subc(.b)	**incd(.b)**		reti
xor(.b)	**dec(.b)**		
	decd(.b)		
	adc(b)		
	sbc(.b)		
	clr(.b)		
	dadc(.b)		
	rla(.b)		
	rlc(.b)		
	tst(.b)		

Example 2: msp430x20x3_P1_01.asm

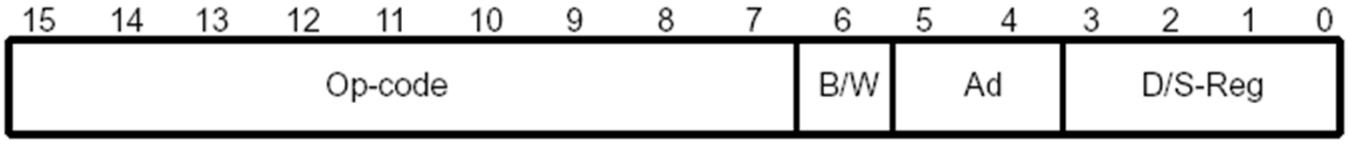
```
;-------------------------------------------------------------------------------
 .cdecls C,LIST,  "msp430x20x3.h"
;-------------------------------------------------------------------------------
            .text                           ; Program Start
;-------------------------------------------------------------------------------
RESET       mov.w   #0280h,SP               ; Initialize stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
SetupP1     bis.b   #001h,&P1DIR            ; P1.0 output
Mainloop    bit.b   #010h,&P1IN             ; P1.4 hi/low?
            jc      ON                      ; jmp--> P1.4 is set
                                            ;
OFF         bic.b   #001h,&P1OUT            ; P1.0 = 0 / LED OFF
            jmp     Mainloop                ;
ON          bis.b   #001h,&P1OUT            ; P1.0 = 1 / LED ON
            jmp     Mainloop                ;
                                            ;
;-------------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET                   ;
            .end
```

Double operand instructions



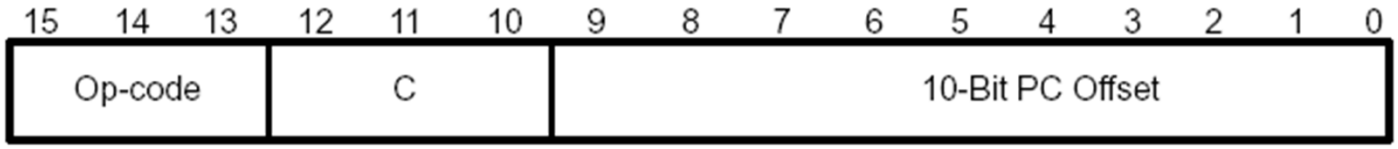
Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
MOV(.B)	src,dst	src → dst	–	–	–	–
ADD(.B)	src,dst	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	src + dst + C → dst	*	*	*	*
SUB(.B)	src,dst	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	dst + .not.src + C → dst	*	*	*	*
CMP(.B)	src,dst	dst – src	*	*	*	*
DADD(.B)	src,dst	src + dst + C → dst (decimally)	*	*	*	*
BIT(.B)	src,dst	src .and. dst	0	*	*	*
BIC(.B)	src,dst	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	src .or. dst → dst	–	–	–	–
XOR(.B)	src,dst	src .xor. dst → dst	*	*	*	*
AND(.B)	src,dst	src .and. dst → dst	0	*	*	*

Single Operand Instruction



Mnemonic	S-Reg, D-Reg	Operation	Status Bits V	N	Z	C
RRC(.B)	dst	C → MSB →.....LSB → C	*	*	*	*
RRA(.B)	dst	MSB → MSB →....LSB → C	0	*	*	*
PUSH(.B)	src	SP – 2 → SP, src → @SP	–	–	–	–
SWPB	dst	Swap bytes	–	–	–	–
CALL	dst	SP – 2 → SP, PC+2 → @SP	–	–	–	–
		dst → PC				
RETI		TOS → SR, SP + 2 → SP	*	*	*	*
		TOS → PC,SP + 2 → SP				
SXT	dst	Bit 7 → Bit 8.....Bit 15	0	*	*	*

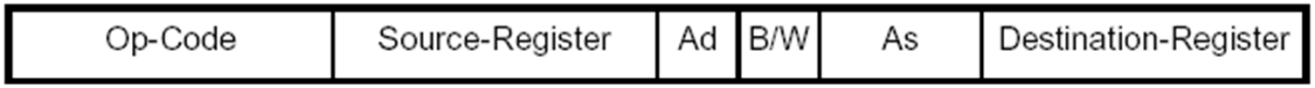
Jump Instructions



Mnemonic	S-Reg, D-Reg	Operation
JEQ/JZ	Label	Jump to label if zero bit is set
JNE/JNZ	Label	Jump to label if zero bit is reset
JC	Label	Jump to label if carry bit is set
JNC	Label	Jump to label if carry bit is reset
JN	Label	Jump to label if negative bit is set
JGE	Label	Jump to label if (N .XOR. V) = 0
JL	Label	Jump to label if (N .XOR. V) = 1
JMP	Label	Jump to label unconditionally

3 Instruction Formats

```
; Format I Source and Destination
```




```
5405            add.w   R4,R5                   ; R4+R5=R5 xxxx
5445            add.b   R4,R5                   ; R4+R5=R5 00xx
```

```
; Format II Destination Only
```


```
6404            rlc.w   R4                      ;
6444            rlc.b   R4                      ;
```

```
; Format III There are 8(Un)conditional Jumps
```


```
3c28            jmp     Loop_1                  ; Goto Loop_1
```

End of lecture 3

Addressing Modes

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register contents are operand
01/1	Indexed mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word.
01/1	Symbolic mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/–	Indirect register mode	@Rn	Rn is used as a pointer to the operand.
11/–	Indirect autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/–	Immediate mode	#N	The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

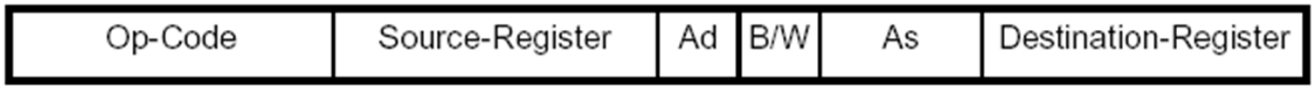
Register Addressing Mode

Op-Code	Source-Register	Ad	B/W	As	Destination-Register
0100	0100	0	0	00	0101

```
4405          mov.w    R4,R5          ;

4445          mov.b    R4,R5          ;
```

Valid for Source and destination As=00, Ad=0
The operand is contained in one of the CPU registers R0 to R15.
This is the fastest addressing mode and needs the least memory .

Register-Indexed Addressing Mode



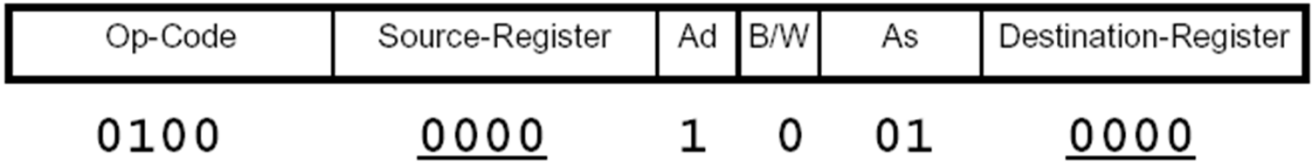

```
4495 01000200   mov.w   100h(R4),200h(R5) ;

44150100        mov.w   100h(R4),R5       ;
```

Valid for Source and destination As=01, Ad=1
The address of the operand is the sum of the index and the contents of the register.

Useful for implementing arrays…above, 100h is the base address of the source array, and the contents of R4 are the index of the source array.

Difference between C and MSP430 assembly: in assembly bytes are indexed, even if we're dealing with an array of words of larger…in C, the index counts the items, whatever size they are

Symbolic Addressing Mode

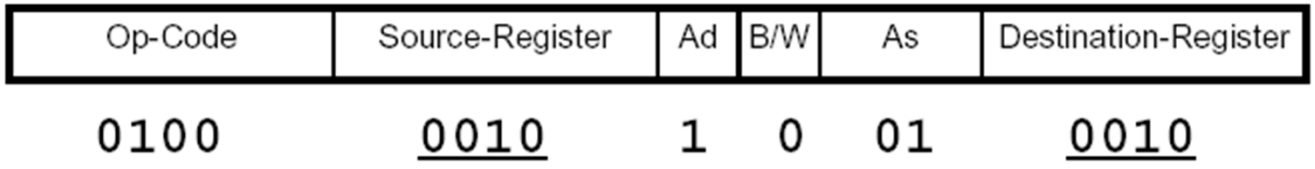



```
4090ffa80006   mov.w    EDE,TONI          ;

4015ffac       mov.w    EDE,R5            ;
```

```
mov.w   LoopCtr, R6     ; load word LoopCtr into R6, symbolic mode
➔ becomes ➔
mov.w   X(PC),R6        ; load word LoopCtr into R6, symbolic mode
Where X = LoopCtr – PC --- calc done by assembler
```

Could also be called “PC relative”
Gives a relative addressing scheme that can be used to create position-independent code. Less useful in flash-based systems than RAM-based (since moving code is very costly in time and energy for flash).

Absolute Addressing Mode




```
4292 01720174   mov.w    &CCR0,&CCR1        ;

42150172        mov.w    &CCR0,R5           ;
```

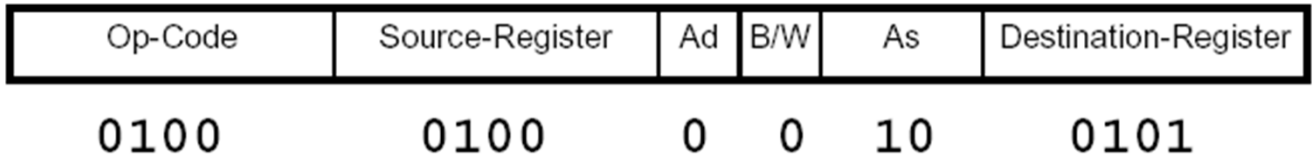
```
mov.b   &P1IN,R6        ; load byte P1IN into R6, abs mode
➔ becomes ➔
mov.b P1IN(SR),R6       ; load byte P1IN into R6, abs mode

SR returns 0 in this case…a kind of constant generator fn
```

SP-relative

- Not claimed as an addressing mode by TI…some other manufacturers call this a distinct addressing mode

```
mov.w   2(SP),R6                ; most recent word but one from stack
```

Register Indirect Addressing Mode




```
4425          mov.w   @R4,R5          ;
4465          mov.b   @R4,R5          ;
```

Memory at ***addr*** held in R4 is moved into R5

This mode can't be used for destination addresses, only source
Instead, need to use indexed addressing for destination:

```
44a50000      mov.w   @R4,0(R5)       ;
```

Register Indirect Autoincrement Addressing Mode

Op-Code	Source-Register	Ad	B/W	As	Destination-Register
0100	0100	0	0	11	0101

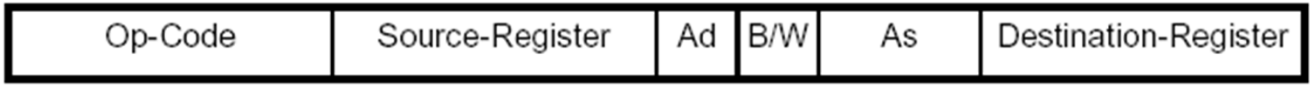
```
4435            mov.w   @R4+,R5             ;

4475            mov.b   @R4+,R5             ;
```

Source only As=11, Ad=n/a
The registers are used as a pointer to the operand. The registers are incremented afterwards - by 1 in byte mode, by 2 in word mode.

Inverse operation:

```
mov.w   R6,0(R5)        ; store word from R6 into addr 0+(R5)=4
incd.w  R5              ; R5 += 2
```

Immediate Addressing Mode




```
0100        0000     0   0   11      0101

4035 1234       mov.w   #1234h,R5           ; Any 16-bit value
```

Source only As=11, Ad=n/a

Any immediate 8 or 16 bit constant can be used with the instruction. The PC is used in autoincrement mode to emulate this addressing mode.

Equivalent to

```
mov.w   @PC+,R6 ; load immediate word 1234h into R6
DW      1234h   ;
```