Assembly Language Programming

EE3376

UTEP

Moving Up Levels of Abstraction

Problems

Algorithms

Language

Machine (ISA) Architecture

Microarchitecture
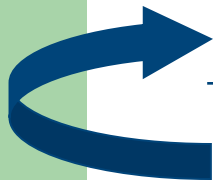
Circuits

Devices

Assembly code

Machine code

MSP430 Architecture

Logic gates, multiplexers, memory, etc.

Transistors

2

High Level vs. Assembly

- High Level Languages
  - More programmer friendly
  - More ISA independent
  - Each high-level statement translates to several instructions in the ISA of the computer

- Assembly Languages
  - Lower level, closer to ISA
  - Very ISA-dependent
  - Each instruction specifies a single ISA instruction
  - Makes low level programming more user friendly
  - More efficient code

3

Assembler Syntax

- Each assembly line begins with either a label, a blank (tab), an asterisk, or a semicolon
- Each line has four fields:

  `{label[:]} mnemonic {operand list} {;comment}`

- Some line examples are:

```
        .sect ".sysmem"  ; data space
var1    .word 2          ; variable var1 declaration
        .text            ; program space
loop:   mov #COUNT,r5    ; get counter
        .end             ; end of program
```

Symbols / Labels

- Symbols
  - Symbols are used as labels, constants, and substitution values
  - Symbols are stored in a symbol table
  - A symbol name
    - is a string of up to 200 alphanumeric characters (A-Z, a-z, 0-9, $, and _)
    - cannot contain embedded blanks
    - first character cannot be a number
    - case sensitive
  - Symbols used as labels become symbolic addresses that are associated with locations in the program
- Label Field
  - Labels are symbols
  - Labels must begin in column 1.
  - A label can optionally be followed by a colon
  - The value of a label is the current value of the Location Counter (address within program)
  - A label on a line by itself is a valid statement
  - Labels used locally within a file must be unique.

5

Mnemonics / Operands

- Mnemonic Field
  - The mnemonic field follows the label field.
  - The mnemonic field cannot start in column 1; if it does, it is interpreted as a label.
  - The mnemonic field contains one of the following items:
    - MSP430 instruction mnemonic (ie. ADD, MOV, JMP)
    - Assembler directive (ie. .data, .list, .equ)
    - Macro directive (ie. .macro, .var, .mexit)
    - Macro call
- Operand Field
  - The operand field follows the mnemonic field and contains one or more operands.
  - The operand field is not required for all instructions or directives.
  - An operand may consist of:
    - Symbols
    - Constants
    - Expressions (combination of constants and symbols)
  - Operands are separated with commas

6

Assembler Directives

- Assembly directives are used to specify:
  - Starting addresses for programs
  - Starting values for memory locations
  - Specify the end of program text.

```
;*********************************************************************************
;    CS/ECEn 124 Example Code
;*********************************************************************************
            .cdecls C,LIST,  "msp430x22x4.h"    ; include C header

COUNT       .equ         2000
;-------------------------------------------------------------------
            .bss         cnt,2                   ; ISR counter
;-------------------------------------------------------------------
            .text                                ; Program reset
RESET:      mov.w        #0x0280,SP              ; Initialize stack pointer
            mov.w        #WDT_MDLY_0_5,&WDTCTL   ; Set Watchdog interval to ~0.5ms
            bis.w        #LPM0+GIE,SR            ; Enter LPM0 w/ interrupt
            jmp          $                       ; Loop forever; interrupts do all

            .sect        ".reset"                ; MSP430 RESET Vector
            .word        RESET                   ; Power Up ISR
            .end
```

Directives

Current Location Counter

Adapted from notes from BYU ECE124

7

Assembly Code Example

```
;*****************************************************************************
;   CS/ECEn 124 Lab 4 - morse.asm: Student Code
;*****************************************************************************
            .cdecls C,LIST,  "msp430x22x4.h"    ; include C header

COUNT       .equ           2000

;----------------------------------------------------------------
            .data                               ; data
            .bss           cnt,2                ; ISR counter
;----------------------------------------------------------------
            .text                               ; Program reset
RESET:      mov.w          #0x0280,SP           ; Initialize stack pointer
            mov.w          #WDT_MDLY_0_5,&WDTCTL ; Set Watchdog interval to ~0.5ms
            mov.b          #WDTIE,&IE1          ; Enable WDT interrupt
            bis.b          #0x01,&P1DIR         ; P1.0 output
            bis.b          #0x20,&P4DIR         ; P4.0 output
            mov.w          #COUNT,&cnt          ; initialize counter
            bis.w          #LPM0+GIE,SR         ; Enter LPM0 w/ interrupt
            jmp            $                    ; Loop forever; interrupts do all

;           Watchdog Timer interrupt service routine
;
WDT_ISR:    xor.b          #0x20,&P4OUT         ; pulse buzzer
            dec.w          &cnt                 ; decrement counter
             jne           WDT_exit
            mov.w          #COUNT,&cnt          ; initialize counter
            xor.b          #0x01,&P1OUT         ; toggle P1.0

WDT_exit:   reti                                ; return from interrupt

            .sect          ".int10"             ; MSP430 RESET Vector
            .word          WDT_ISR              ; Watchdog ISR

            .sect          ".reset"             ; MSP430 RESET Vector
            .word          RESET                ; Power Up ISR
            .end
```

Instructions

Labels

Comments

Directives

8

Common Assembler Directives

| Mnemonic and Syntax	Description
.bss *symbol, size in bytes*[*, alignment*]	Reserves *size* bytes in the .bss (uninitialized data) section
.sect "*section name*"	Assembles into a named (initialized) section
.text	Assembles into the .text (executable code) section
.byte *value$_1$*[*, ..., value$_n$*]	Initializes one or more successive bytes in the current section
.string "*string$_1$*"[*, ..., "string$_n$*"]	Initializes one or more text strings
.word *value$_1$*[*, ... , value$_n$*]	Initializes one or more 16-bit integers
.align [*size in bytes*]	Aligns the LC on a boundary specified by *size in bytes;* must be a power of 2; defaults to word (2 byte)
.def *symbol$_1$*[*, ... , symbol$_n$*]	Identifies one or more symbols that are defined in current module and that can be used in other modules
.include ["]*filename*["]	Includes source statements from another file
.ref *symbol$_1$*[*, ... , symbol$_n$*]	Identifies one or more symbols used in the current module that are defined in another module
symbol .equ *value*	Equates *value* with *symbol*
symbol .set *value*	Equates *value* with *symbol*
.cdecls [*options,*] "*filename*"	Share C headers between C and assembly code
.end	Ends program

CCS Window – C/C++ Perspective

Code Window
- Breakpoints
- Syntax highlighting

Independent debugging and Programming view

1-click project debug

Project View
- List of Projects

Problems View
- Information
- Warnings
- Errors

Console
Build information
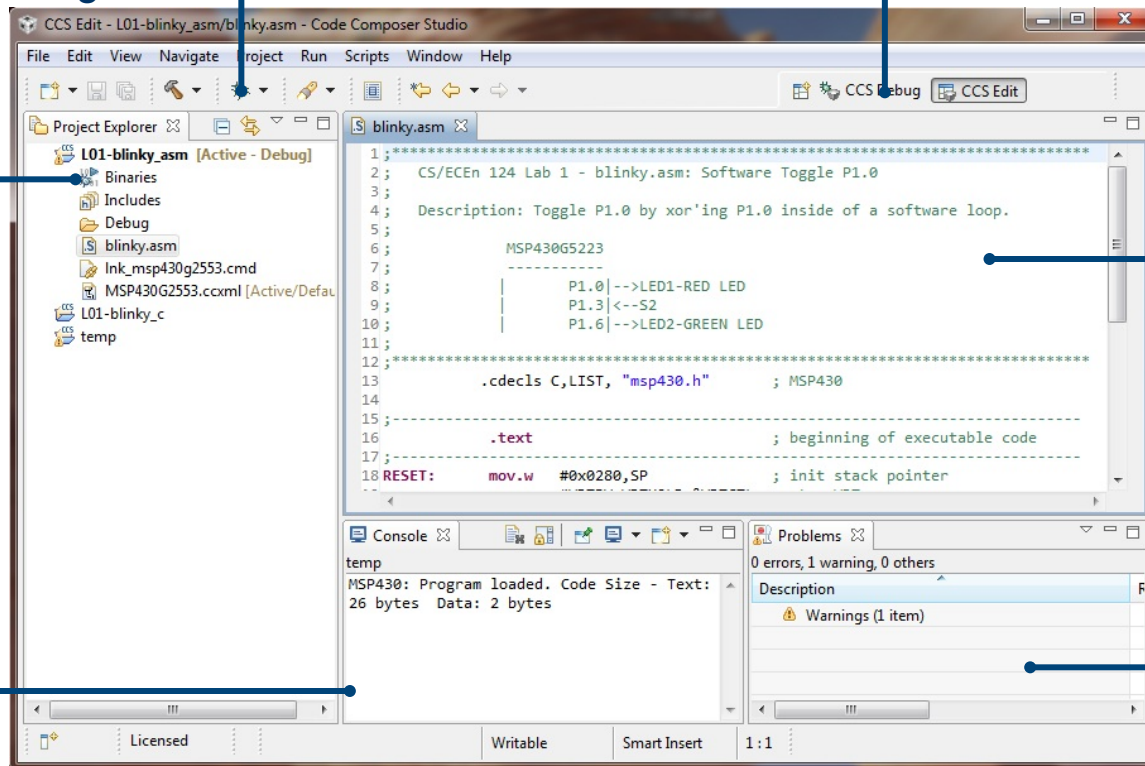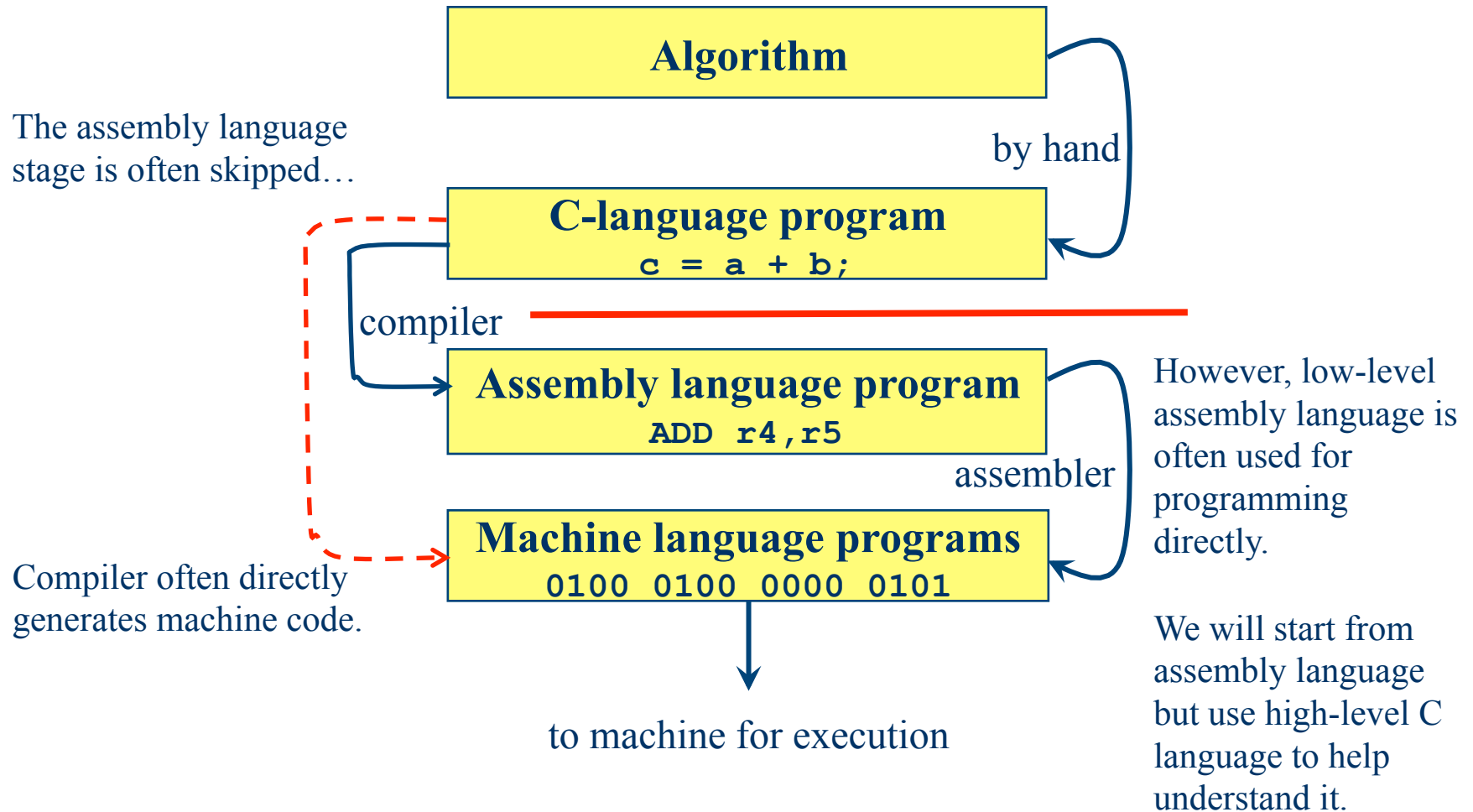
Assembly List File

- A line in a listing file has four fields:
  - Field 1: contains the source code line counter
  - Field 2: contains the section program counter
  - Field 3: contains the object code
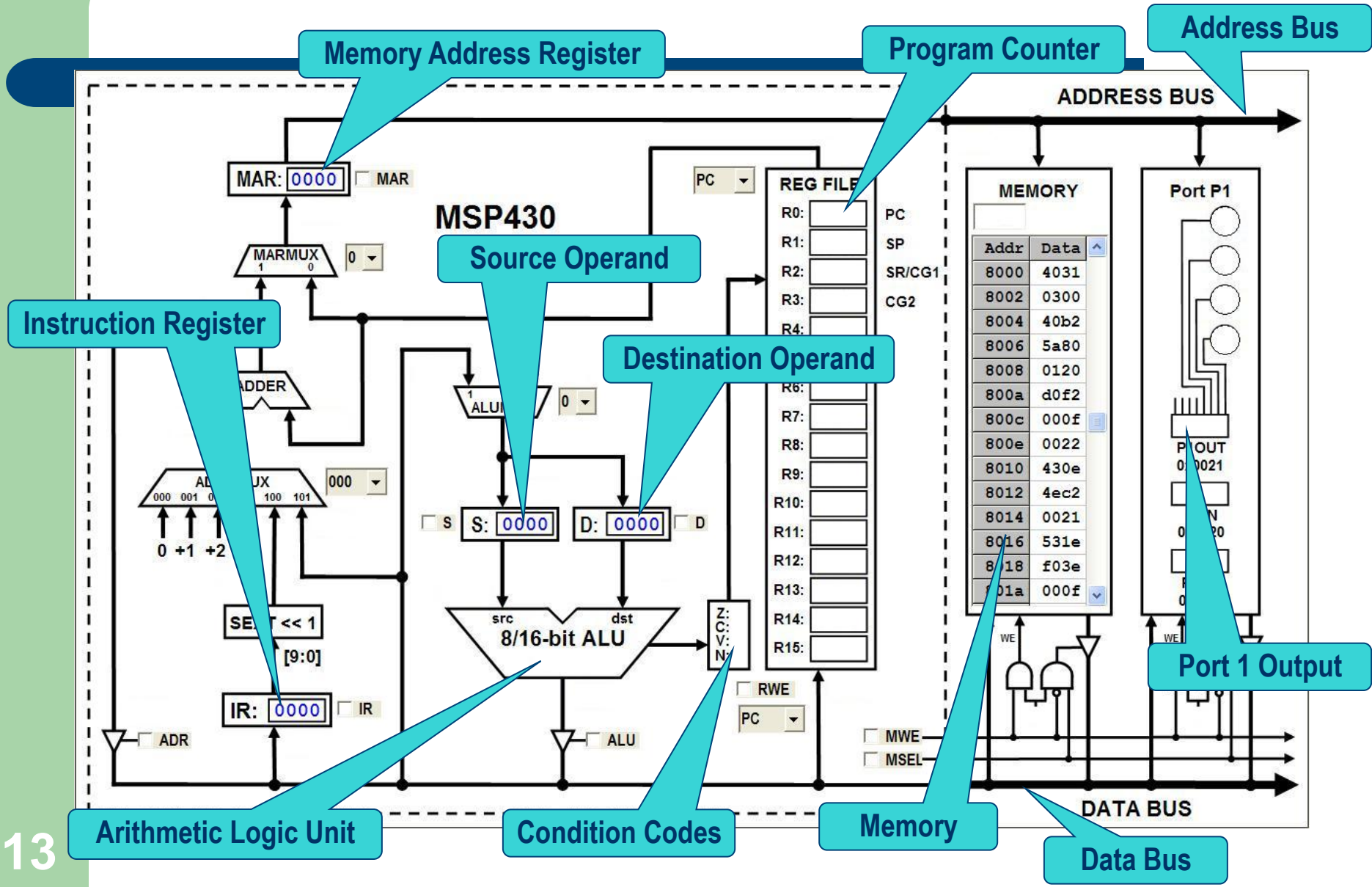 - Field 4: contains the original source statement.

```
26                 ***********************************************
27                 * Define another section for more variables. *
28                 ***********************************************
29 0000            var2     .usect   "newvars", 1
30 0001            inbuf    .usect   "newvars", 7
31                 ***********************************************
32                 *        Assemble more code into .text.       *
33                 ***********************************************
34 0008                     .text
35 0008 403C       mpy:     MOV.W    #0x3456,R12
   000a 3456
36 000c 421D                MOV.W    &coeff,R13
   000e 0000!
37 0010 1290                CALL     _mpyi
   0012 FFEE!
38                 ***********************************************
39                 * Define a named section for int. vectors. *
40                 ***********************************************
41 0000                     .sect    "vectors"
42 0000 0300                .word    0x300
```

Field 1 Field 2 Field 2   Field 4

Compilation

Algorithm

by hand

The assembly language stage is often skipped…

C-language program
`c = a + b;`

compiler

Assembly language program
`ADD r4,r5`

However, low-level assembly language is often used for programming directly.

assembler

Compiler often directly generates machine code.

Machine language programs
`0100 0100 0000 0101`

to machine for execution

We will start from assembly language but use high-level C language to help understand it.

MSP 430 Micro-Architecture

Adapted from notes from BYU ECE124

MSP 430 Data Storage

- The MSP430 CPU has 64KB memory space and 16 registers for data storage

- R0 (PC) – Program Counter
  - This register always points to the next instruction to be fetched

- R1 (SP) – Stack Pointer
  - The MSP430 CPU stores the return address of routines or interrupts on the stack
  - User programs store local data on the stack

- R2 (SR/CG1) – Status Register
  - The status of the MSP430 CPU is defined by a set of bits contained in register R2

14

MSP 430 Registers

- R2 (SR/CG1), R3 (CG2) – Constant Generators
  - Six different constants commonly used in programming can be generated using the registers R2 and R3, without adding a 16-bit extension word of code to the instruction

Register	As	Constant	Remarks
R2	00	-	Register mode
R2	01	(0)	Absolute 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

15

MSP 430 Registers

- R4-R15 – General Purpose registers
    - The general purpose registers R4 to R15 can be used as data registers, data pointers and indices.
    - They can be accessed either as a byte or as a word
    - Instruction formats support byte or word accesses
    - The status bits of the CPU in the SR are updated after the execution of a register instruction.

16

MSP430G2553 Memory Map

Format I: 12 Double Operand Instructions

- Double operand instructions:

Mnemonic	Operation	Description
Arithmetic instructions		
ADD(.B or .W) src,dst	src+dst→dst	Add source to destination
ADDC(.B or .W) src,dst	src+dst+C→dst	Add source and carry to destination
DADD(.B or .W) src,dst	src+dst+C→dst (dec)	Decimal add source and carry to destination
SUB(.B or .W) src,dst	dst+.not.src+1→dst	Subtract source from destination
SUBC(.B or .W) src,dst	dst+.not.src+C→dst	Subtract source and not carry from destination
Logical and register control instructions		
AND(.B or .W) src,dst	src.and.dst→dst	AND source with destination
BIC(.B or .W) src,dst	.not.src.and.dst→dst	Clear bits in destination
BIS(.B or .W) src,dst	src.or.dst→dst	Set bits in destination
BIT(.B or .W) src,dst	src.and.dst	Test bits in destination
XOR(.B or .W) src,dst	src.xor.dst→dst	XOR source with destination
Data instructions		
CMP(.B or .W) src,dst	dst-src	Compare source to destination
MOV(.B or .W) src,dst	src→dst	Move source to destination

18

Adapted from notes from BYU ECE124

Examples

```
MOV.w    #0x08,R5           ; move source to destination
                            ; assign a hexadecimal value 0x08 to Register R5

AND.w    #0x00,R6           ; bitwise AND source with destination
                            ; whatever value in R6 is ANDed with 0 -> R6=0

ADD.w    #0x03,R6           ; add source to destination
                            ; R6 = R6+3 = 0+3 = 3

SUB.w    R6, R5             ; subtract source from destination
                            ; R5 = R5-R6 = R5+(Not R6)+1 = 8-3 = 5

XOR.w    R6, R5             ; bitwise XOR source with destination
                            ; R5 = 0011 XOR 0101 = 0110 = 6

BIC.w    #0x03, R5          ; clear bits in destination
                            ; (Not 0011) AND 0110 = 1100 AND 0110 = 0100 = 4

BIS.w    #0x08, R5          ; set bits in destination
                            ; 1000 OR 0100 = 1100 = 12

BIT.w    #0x08, R5          ; test bits in destination
                            ; 1000 AND 1100 = 1000 -> Bit 3 is not zero

CMP.w    R6, R5             ; compare source to destination
                            ; R5-R6 = 12-6 = 6 greater than 0, so R5 > R6
```

19

Format II: 7 Single Operand Instructions

- ● Single operand instructions:

Mnemonic	Operation	Description
Logical and register control instructions		
`RRA(.B or .W) dst`	MSB→MSB→… LSB→C	Roll destination right
`RRC(.B or .W) dst`	C→MSB→…LSB→C	Roll destination right through carry
`SWPB( or .W) dst`	Swap bytes	Swap bytes in destination
`SXT dst`	bit 7→bit 8…bit 15	Sign extend destination
`PUSH(.B or .W) src`	SP-2→SP, src→@SP	Push source on stack
Program flow control instructions		
`CALL(.B or .W) dst`	SP-2→SP, PC+2→@SP dst→PC	Subroutine call to destination
`RETI`	@SP+→SR, @SP+→SP	Return from interrupt

Examples

```
MOV.w   #0xF009,R5          ; move source to destination
                            ; assign a hexadecimal value 0x08 to Register R5


RRA.w   R5                  ; Roll destination right and send LSB to Carry
                            ; 1111 0000 0000 1001 -> 1111 1000 0000 0100  C=1

RRC.w   R5                  ; Roll destination right through Carry
                            ; 1111 1000 0000 0100 -> 1111 1100 0000 0010  C=0

SWPB.w  R5                  ; subtract source from destination
                            ; 1111 1100 0000 0010 -> 0000 0010 1111 1100

SXT     R5                  ; sign extension
                            ; 1111 1100 -> 1111 1111 1111 1100
                            ; (bit 7 is 1) bits 8~15 are all set to 1

PUSH, CALL, and RETI will be discussed later
```

21

Jump Instruction Format

- Jump instructions are used to direct program flow to another part of the program.

- The condition on which a jump occurs depends on the Condition field consisting of 3 bits:
  - JNE/JNZ : jump if not equal
  - JEQ/JZ : jump if equal
  - JNC/JLO : jump if carry flag equal to zero
  - JC/JHS : jump if carry flag equal to one
  - JN : jump if negative (N = 1)
  - JGE : jump if greater than or equal (N = V)
  - JL : jump if lower (less) (N ≠ V)
  - JMP : unconditional jump (no condition check)

22

Examples

```
MOV.w    #0x05,R5            ; move source to destination
                            ; assign a hexadecimal value 0x05 to Register R5

MOV.w    #0x03,R6            ; move source to destination
                            ; assign a hexadecimal value 0x03 to Register R6

CMP.w    R6, R5             ; compare source to destination
                            ; R5-R6 = 5-3 = 2 greater than 0, so R5 > R6


JNE      somewhere          ; jump if not equal
                    ; The program will jump to "somewhere" because R5 ≠ R6
```

23

Emulated Instructions

- In addition to the 27 instructions of the CPU there are 24 emulated instructions

- The CPU coding is unique

- The emulated instructions make reading and writing code easier, but do not have their own op-codes

- Emulated instructions are replaced automatically by CPU instructions by the assembler

- There are no penalties for using emulated instructions.

24

Emulated Instructions

Mnemonic	Operation	Emulation	Description
Arithmetic instructions			
ADC(.B or .W) dst	dst+C→dst	ADDC(.B or .W) #0,dst	Add carry to destination
DADC(.B or .W) dst	dst + C → dst (decimally)	DADD(.B or .W) #0,dst	Decimal add carry to destination
DEC(.B or .W) dst	dst-1→dst	SUB(.B or .W) #1,dst	Decrement destination
DECD(.B or .W) dst	dst-2→dst	SUB(.B or .W) #2,dst	Decrement destination twice
INC(.B or .W) dst	dst+1→dst	ADD(.B or .W) #1,dst	Increment destination
INCD(.B or .W) dst	dst+2→dst	ADD(.B or .W) #2,dst	Increment destination twice
SBC(.B or .W) dst	dst+0FFFFh+C→dst dst+0FFh→dst	SUBC(.B or .W) #0,dst	Subtract source and borrow /.NOT. carry from dest.

25

Emulated Instructions

Mnemonic	Operation	Emulation	Description
Logical and register control instructions			
INV(.B or .W) dst	.NOT.dst→dst	XOR(.B or .W) #0(FF)FFh,dst	Invert bits in destination
RLA(.B or .W) dst	C←MSB←MSB-1 LSB+1←LSB←0	ADD(.B or .W) dst,dst	Rotate left arithmetically (multiplied by 2)
RLC(.B or .W) dst	C←MSB←MSB-1 LSB+1←LSB←C	ADDC(.B or .W) dst,dst	Rotate left through carry
Program flow control			
BR dst	dst→PC	MOV dst,PC	Branch to destination
DINT	0→GIE	BIC #8,SR	Disable (general) interrupts
EINT	1→GIE	BIS #8,SR	Enable (general) interrupts
NOP	None	MOV #0,R3	No operation
RET	@SP→PC SP+2→SP	MOV @SP+,PC	Return from subroutine

26

Emulated Instructions

Mnemonic	Operation	Emulation	Description
Data instructions			
CLR(.B or .W) dst	0→dst	MOV(.B or .W) #0,dst	Clear destination
CLRC	0→C	BIC #1,SR	Clear carry flag
CLRN	0→N	BIC #4,SR	Clear negative flag
CLRZ	0→Z	BIC #2,SR	Clear zero flag
POP(.B or .W) dst	@SP→temp SP+2→SP temp→dst	MOV(.B or .W) @SP +,dst	Pop byte/word from stack to destination
SETC	1→C	BIS #1,SR	Set carry flag
SETN	1→N	BIS #4,SR	Set negative flag
SETZ	1→Z	BIS #2,SR	Set zero flag
TST(.B or .W) dst	dst + 0FFFFh + 1 dst + 0FFh + 1	CMP(.B or .W) #0,dst	Test destination

27

Example: Emulated Instructions

- Emulated instructions are replaced automatically by CPU instructions by the assembler
- Could be replaced differently
- Clear the contents of register R5:

```
CLR R5 = MOV.W #0, R5
```

- Increment the content of register R5:

```
INC R5 = ADD.W #1, R5
```

- Decmrent the content of register R5:

```
DEC R5 = SUB.W #1, R5
```

28

Example: Emulated Instructions

- Decrement by two the contents of register R5:

  `DECD R5 = SUB.W #2, R5`

- Do not carry out any operation:

  `NOP = MOV.W R3, R3`

- Add the carry flag to the register R5:

  `ADC R5 = ADC.W #0, R5`

29

Source Addressing Modes

- The MSP430 has four basic modes for the source address:
  - **Rs** - Register
  - **x(Rs)** - Indexed Register
  - **@Rs** - Register Indirect (not for destination)
  - **@Rs+** - Indirect Auto-increment (not for destination)
- In combination with registers R0-R3, three additional source addressing modes are available:
  - **label** - PC Relative, **x(PC)**
  - **&label** – Absolute, **x(SR)**
  - **#n** – Immediate, **@PC+** (not for destination)

30

Destination Addressing Modes

- There are two basic modes for the destination address:
  - **Rd** - Register
  - **x(Rd)** - Indexed Register

- In combination with registers R0/R2, two additional destination addressing modes are available:
  - **label** - PC Relative, **x(PC)**
  - **&label** – Absolute, **x(SR)**

Register Mode (Rn)

- The most straightforward addressing mode and is available for both source and destination
  - **Example:**

    mov.w r5,r6 ; move word from r5 to r6

- The registers are specified in the instruction; no further data is needed

- Also the fastest mode and does not require an addition cycle

- Byte instructions use only the lower byte, but clear the upper byte when writing

0	1	0	0	0	1	0	1	0	0	0	0	0	1	1	0
Op-code				S-reg				Ad	b/w	As		D-reg			
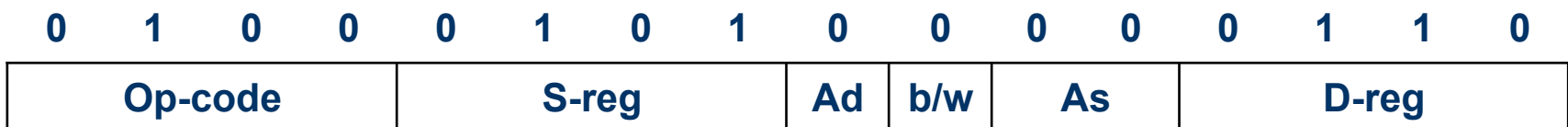
Indexed Mode x(Rn)

- The address is formed by adding a constant (index) to the contents of a CPU register

  - **Example:**

    mov.b 3(r5),r6   ; move byte from

    ;   M(3_{10}+r5) to r6

- Indexed addressing can be used for source and/or destination, value in r5 is unchanged.

- The index is located in the memory word following the instruction and requires an additional memory cycle

- There is no restriction on the address for a byte, but words must lie on even addresses

0	1	0	0	0	1	0	1	0	1	0	1	0	1	1	0
Op-code				S-reg				Ad	b/w	As		D-reg			

Symbolic Mode (PC Relative)

- The address if formed by adding a constant (index) to the program counter (PC)
  - **Example:** (mov.w x(PC), r6 where x=Cnt-PC)

    mov.w Cnt,r6   **; move word**
                                    ; M(Cnt) or M(x+PC) to r6

- The PC relative index is calculated by the assembler

- Produces position-independent code, but rarely used in the MSP430 because absolute addressing can reach all memory addresses

- Note: this is NOT an appropriate mode of addressing when referencing fixed locations in memory such as the special function registers (SFR's)

0	1	0	0	0	0	0	0	0	0	0	1	0	1	1	0
Op-code				S-reg				Ad	b/w	As		D-reg			

34

Absolute Mode (&label)

- The address is formed directly from a constant (index) and specified by preceding a label with an ampersand (&)
  - **Example:** (mov.w x(SR), r6 where 0 is used for SR)

mov.w &Cnt,r6   ; move word
                       ; M(Cnt) to r6

- Same as indexed mode with the base register value of 0 (by using the status register SR as the base register)
- The absolute address is stored in the memory word following the instruction and requires an additional cycle
- Note: this is the preferred mode of addressing when referencing fixed locations in memory such as the special function registers (SFR's)

0	1	0	0	0	0	1	0	0	0	0	1	0	1	1	0
Op-code				S-reg				Ad	b/w	As		D-reg			

Indirect Register Mode (@Rn)

- The address of the operand is formed from the contents of the specified register
  - **Example:**

mov.w @r5,r6   ; move word
; M(r5) to r6

- Only available for source operands
- Same as indexed mode with index equal to 0, but does not require an additional instruction word
- The value of the indirect register is unchanged

0	1	0	0	0	1	0	1	0	0	1	0	0	1	1	0
Op-code				S-reg				Ad	b/w	As		D-reg			

Indirect Autoincrement Mode (@Rn+)

- The address of the operand is formed from the contents of the specified register and afterwards, the register is automatically increment by 1 if a byte is fetched or by 2 if a word is fetched
  – **Example:**

  mov.w @r5+,r6   **; move word**
                                      ; M(r5) to r6
                                      ; increment r5 by 2

- Only available for source operands.
- Usually called post-increment addressing.
- **Note: All operations on the first address are fully completed before the second address is evaluated**

0	1	0	0	0	1	0	1	0	0	1	1	0	1	1	0
Op-code				S-reg				Ad	b/w	As		D-reg			

37

Immediate Mode (#n)

- The operand is an immediate value
  - **Example** (mov.w @PC+, r6)
    ### mov.w #100,r6   ; 100 -> r6
- The immediate value is located in the memory word following the instruction
- Only available for source operands
- The immediate mode of addressing is a special case of auto-increment addressing that uses the program counter (PC) as the source register.
- The PC is automatically incremented after the instruction is fetched; hence points to the following word

0	1	0	0	0	0	0	0	0	0	1	1	0	1	1	0
Op-code				S-reg				Ad	b/w	As		D-reg			

38

Examples - Source

R5	0010
R6	0020
R7	9000
R8	9004
R9	9006

registers

	memory
x9000	000F
x9002	0010
x9004	0011
x9006	0012
x9008	0013

label:

```
mov.w R5, R6          ; move the content of R5 (0010) to R6
mov.w @R7, R8         ; use the content of R7 (9000) as the address to move data (000F) to R8
mov.w &0x9004, R6     ; go to the absolute address (9004) to move data (0011) to R6
mov.w 2(R7), R9       ; use the content of R7 (9000) as the base address and offset it by 2 (9002)
                                  to move data (0010) to R9
mov.w &0x9006, R5     ; go to the absolute address (9006) to move data (0012) to R5
mov.w @R7+, R6        ; use the content of R7 (9000) as the address to move data (000F) to R6 and
                                  then increment R7 by 2 (one word is 2 bytes)
mov.w #0x9000, R7     ; move the immediate value (9000) to R7
mov.w label, R7       ; move the data (0012) in the memory space represented by "label" to R7
```

39

Examples - Destination

registers   memory

R5	0010
R6	0020
R7	9000
R8	9004
R9	9006

x9000	000F
x9002	0010
x9004	0011
x9006	0012
x9008	0013

label: at x9006

mov.w R5, R6   ; move the content of R5 (0010) to R6

mov.w R5, 4(R7)   ; use the content of R7 (9000) as the base address and offset it by 4 (9004)
                     and move the content of R5 (0010) to the effective address (9004)

mov.w R6, label   ; move the content of R6 (0012 moved from R5) to the memory space
                     represented by "label"

mov.w &0x9004, &0x9008   ; go to the absolute address (9004) to move data (0011) to the
                     memory location of the absolute address (9008)

mov.w R6, &label   ; move the content of R6 (0012 moved from R5) to the memory space
                     represented by the absolute address "label"

40

High Level vs. Assembly

- High Level Languages
  – More programmer friendly
  – More ISA independent
  – Each high-level statement translates to several instructions in the ISA of the computer
- Assembly Languages
  – Lower level, closer to ISA
  – Very ISA-dependent
  – Each instruction specifies a single ISA instruction
  – Makes low level programming more user friendly
  – More efficient code

41

The first assembly program

```
;****************************************************************************
;     MSP430 Micro-Architecture Simulator Code;
;     Description:
;        Display an incrementing counter in the simulator LEDs.
;                      MSP430
;               -----------------
;               |               |
;               |         P1.0|-->RED LED
;               |         P1.1|-->YELLOW LED
;               |         P1.2|-->BLUE LED
;               |         P1.3|-->GREEN LED
;****************************************************************************
                .cdecls  C,LIST,  "msp430x22x4.h"
;---------------------------------------------------------------------------
                .text                          ; Program reset
;---------------------------------------------------------------------------
RESET:
                mov.w    #0x0300,SP             ; Initialize stack pointer
                mov.w    #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
                bis.b    #0x0f,&P1DIR           ; Set P1.0-3 output
                mov.w    #0,r14
Mainloop:       mov.b    r14,&P1OUT             ; output P1.0-3
                inc.w    r14
                and.w    #0x000f,r14            ; mask counterWait:
                mov.w    Delay,r15              ; Delay to R15
                push     r15
L1:             dec.w    0(sp)                  ; decrement delay counter
                jnz      L1                     ; delay over?
                mov.w    @sp+,r15               ; Y
                jmp      Mainloop               ; repeat
Delay:          .word    2
;---------------------------------------------------------------------------
                .sect    ".reset"               ; MSP430 RESET Vector
                .short   RESET                  ; NMI
                .end
```

Output Pins to control the LEDs

Comments

Instructions

Directives

Labels

Comments

Directives

42

Adapted from notes from BYU ECE124

What does it do?

Move an immediate value #0x0300 into stack pointer register SP (R1)

Move an immediate value that is the OR'ed result of WDTPW and WDTHOLD (both are predefined memory addresses) into a memory location at the absolute address WDTCTL (also a predefined address)

Set the lowest 4 bits of the value stored in the absolute address location P1DIR to all '1'. This is to set Pins 0~3 of Port 1 to be for output controls (turn LEDs on or off)

Move an immediate value 0 to register #14. 0 is actually created by hardware

Move the value stored in register #14 to the absolute address location PIOUT (predefined for Port 1 output register)

Increment value in register #14 by 1

AND the value in register #14 with 0x000F (0000 0000 0000 1111) to keep only the lowest 4 bits in r14.

Move a value "Delay" to r15

Move the value in r15 onto the stack

.word directive assigned a work value 2 to Delay

Unconditionally (always) jump to location labeled Mainloop

Move the value stored on top of the stack to r15 and then increment SP by 2

If the previous instruction result is NOT zero, then jump to location labeled L1, else continue to the next instruction

Decrement the value stored on top of the stack

```
RESET:
          mov.w     #0x0300,SP           ; Initialize stack pointer
          mov.w     #WDTPW+WDTHOLD,&WDTCTL ; Sto...
          bis.b     #0x0f,&P1DIR          ; Set P1.0-3 output
          mov.w     #0,r14
Mainloop: mov.b     r14,&P1OUT
          inc.w     r14
          and.w     #0x000f,r14           ; Mask counterWait...
          mov.w     Delay,r15             ; Delay to R15
          push      r15
L1:       dec.w     0(sp)                 ; decrement delay
          jnz       L1                    ; delay over?
          mov.w     @sp+,r15              ; y
          jmp       Mainloop              ; repeat
Delay:    .word     2
```

43

Adapted from notes from BYU ECE124

What does it do?

Immediate mode

Register mode

Absolute mode

```
RESET:
        mov.w    #0x0300,SP              ; Initialize stack pointer
        mov.w    #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
        bis.b    #0x0f,&P1DIR            ; Set P1.0-3 output
        mov.w    #0,r14
Mainloop: mov.b  r14,&P1OUT             ; output P1.0-3
        inc.w    r14
        and.w    #0x000f,r14            ; mask counterWait:
        mov.w    Delay,r15              ; Delay to R15
        push     r15
L1:     dec.w    0(sp)                  ; decrement delay counter
        jnz      L1                     ; delay over?
        mov.w    @sp+,r15               ; y
        jmp      Mainloop               ; repeat
Delay:  .wo      2
```

Label (PC relative) mode

Indexed register mode

Indirect auto-increment mode

44

Adapted from notes from BYU ECE124

Three Basic Constructs
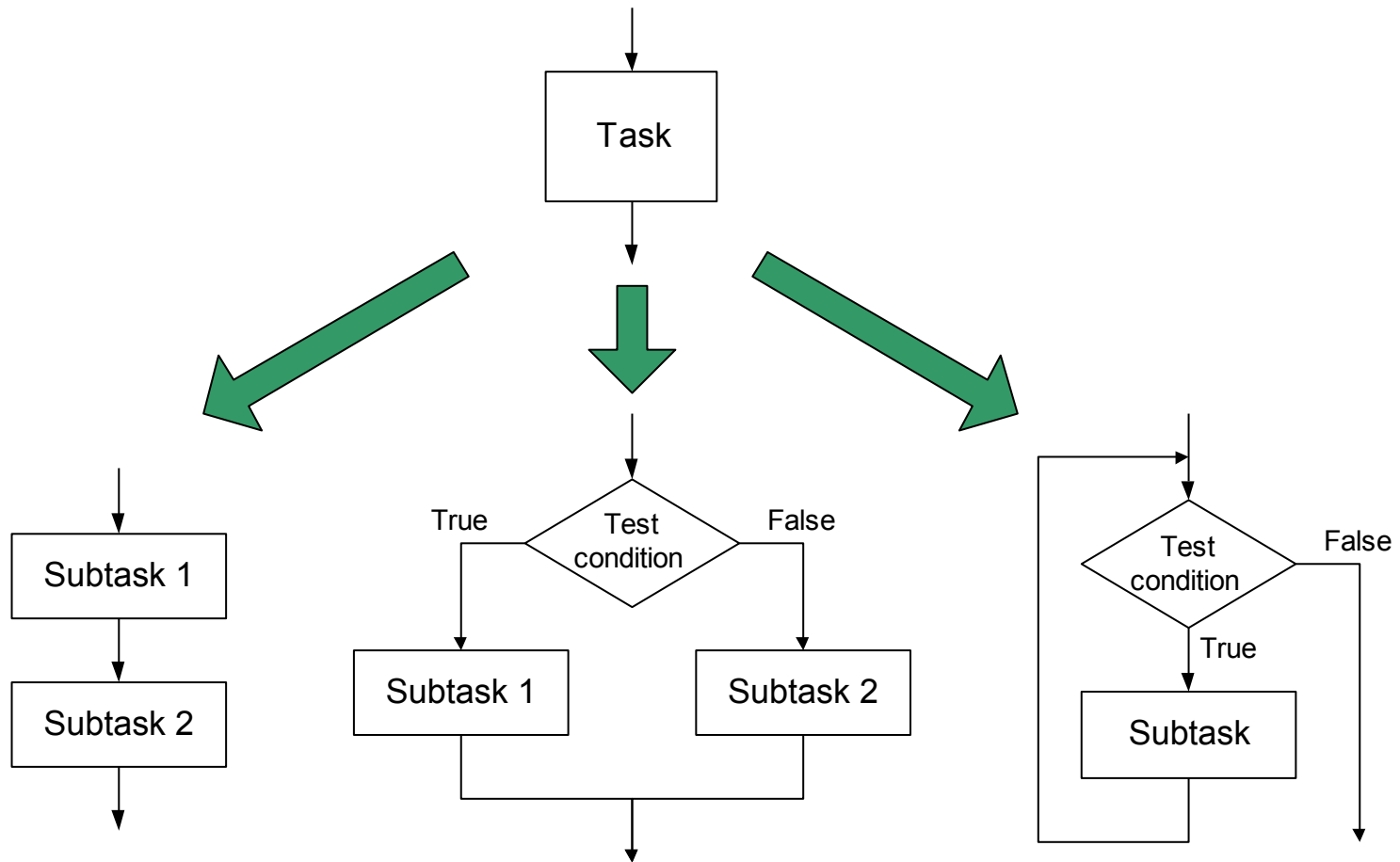
Sequential   **Conditional**   **Iterative**

45

if-then-else Translation

```
        cmp.w #1,&buzzerON      ;      if (buzzerON == 1)
        jne myElse              ;      {
        xor.b #0x20,&P4OUT      ;        pulse_buzzer();
        bis.b #0x02,&P1OUT      ;        turn_on_LED();
        jmp   myNext            ;      }
myElse:                         ;      else
        bic.b #0x02,&P1OUT      ;      {
                                ;        turn_off_LED();
myNext:                         ;      }
```


Adapted from notes from BYU ECE124

46
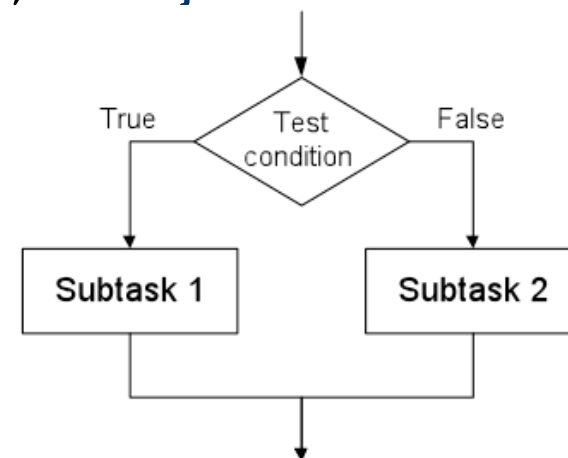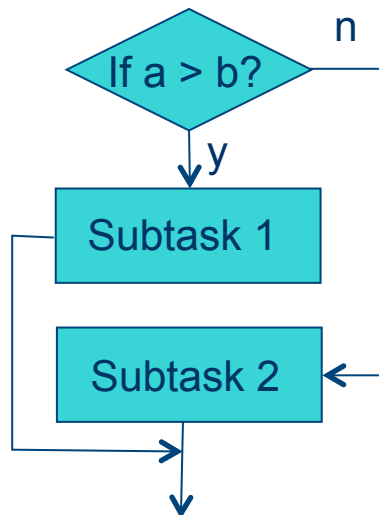
while Translation

```
        while_loop:              ; #define TRUE 1
            bit.w #1,R4          ;
            jnz while_done       ; while (TRUE)
            call  #LED_ON        ; {
            call  #delay         ;    LED_ON();
            call  #LED_OFF       ;    delay();
            call  #delay         ;    LED_OFF();
            jmp   while_loop     ;    delay();
                                 ; }
        while_done:              ;
```
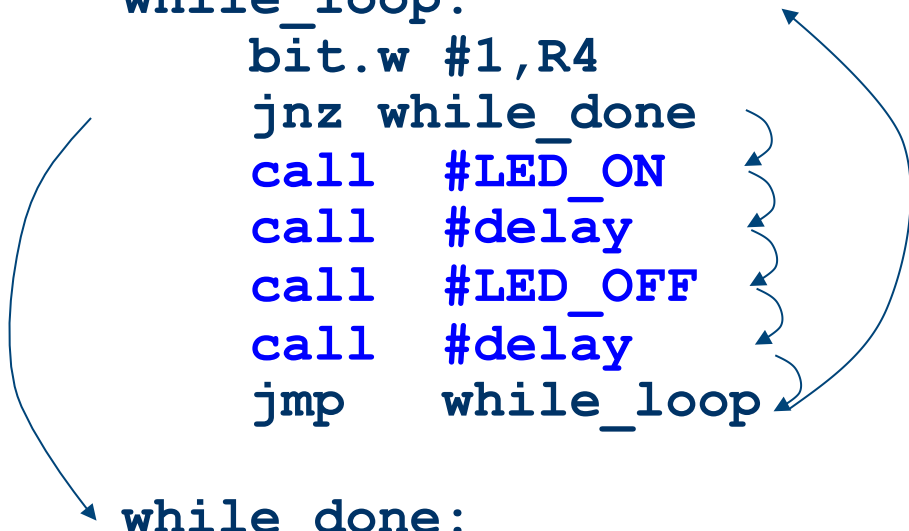
Test
condition

False

True

Subtask

47
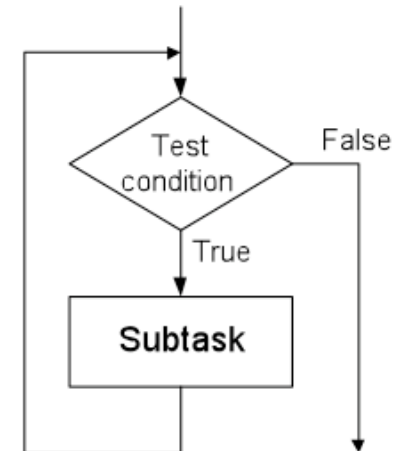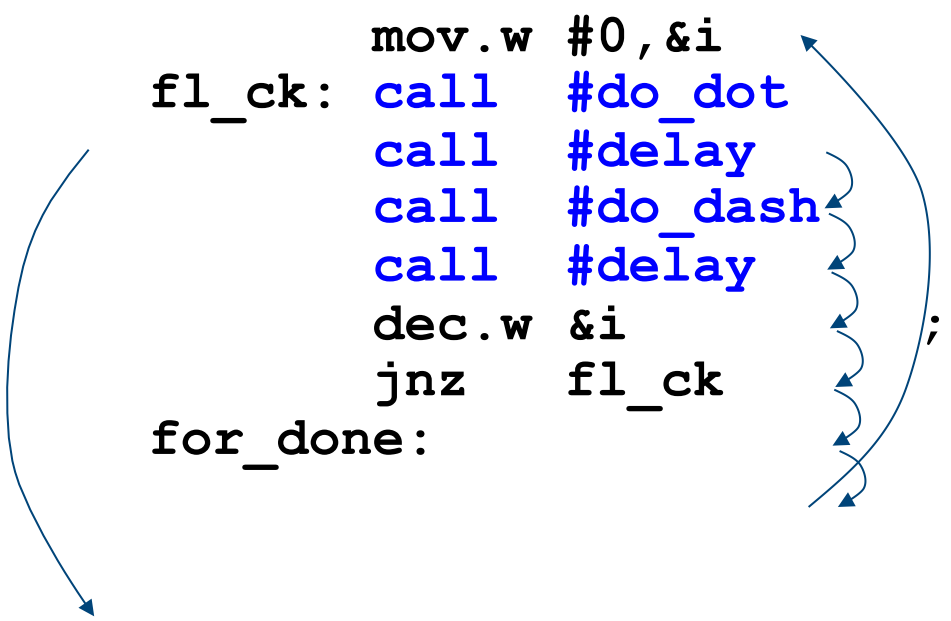
for-loop Translation

```
        .bss  i,2           ;  int i;

        mov.w #0,&i         ;  for(i=0; i<10; i++)
fl_ck:  cmp.w #10,&i        ;  {
        jge for_done        ;
        call  #do_dot       ;     do_dot();
        call  #delay        ;     delay();
        call  #do_dash      ;     do_dash();
        call  #delay        ;     delay();
        add.w #1,&i         ;
        jmp   fl_ck         ;  }

for_done:                   ;
```

48

Better for-loop Translation

```
        .bss   i,2              ;  int i;

        mov.w #0,&i             ;  for(i=10; i>0; i--)
fl_ck:  call  #do_dot           ;  {  do_dot();
        call  #delay            ;     delay();
        call  #do_dash          ;     do_dash();
        call  #delay            ;     delay();
        dec.w &i                ;
        jnz   fl_ck             ;  }
for_done:                       ;
```

switch/case Translation

- ## switch/case

```
        cmp.w #DOT,&myByte          ;    switch (myByte)
        jne sw_01                   ;    {
        call   #do_dot              ;      case DOT:
        jmp    sw_end               ;        do_dot();
                                    ;        break;
    sw_01:
        cmp.w #DASH,&myByte         ;      case DASH:
        jne sw_end                  ;        do_dash();
        call   #do_dash             ;        break;
        jmp    sw_end               ;
                                    ;      default:
    sw_end:                         ;    }
```
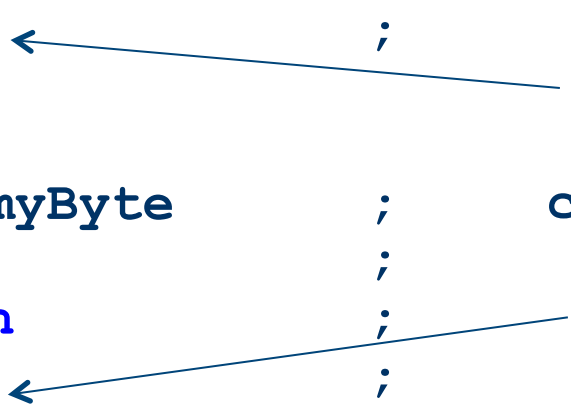
Cycles Per Instruction...

Src	Dst	Cycles	Length	Example
Rn	Rm	1	1	MOV R5,R8
	@Rm	2	1	MOV R5,@R6
	x(Rm)	4	2	ADD R5,4(R6)
	EDE	4	2	XOR R8,EDE
	&EDE	4	2	MOV R5,&EDE
#n	x(Rm)	5	3	MOV #100,TAB(R8)
&TONI	&EDE	6	3	MOV &TONI,&EDE

See "**How To Determine Cycles Per Instruction...**" in Blinky Lab instructions.

51

(a) Two operands (Format I), destination is register.

add.w Rs, Rd

fetch instruction

add.w @Rs, Rd

fetch instruction
fetch source @Rs

Include #N & @Rs+

add.w S(Rs), Rd

fetch instruction
fetch S
fetch source S(Rs)

Include Symbolic & Absolute Address

(b) Two operands (Format I), destination is indexed.

add.w Rs, D(Rd)

fetch instruction
fetch D
fetch destination D(Rd)
write destination D(Rd)

Include Symbolic & Absolute Address

add.w @Rs, D(Rd)

fetch instruction
fetch source @Rs
fetch D
fetch destination D(Rd)
write destination D(Rd)

Include Symbolic & Absolute Address

add.w S(Rs), D(Rd)

fetch instruction
fetch S
fetch source S(Rs)
fetch D
fetch destination D(Rd)
write destination D(Rd)

Include Symbolic & Absolute Address

(c) One operand (Format II)

rra.w Rs

fetch instruction

rra.w @Rs

fetch instruction
fetch source @Rs
write source @Rs

rra.w S(Rs)

fetch instruction
fetch S
fetch source S(Rs)
write source S(Rs)

Instruction Timing

```
mainloop:    xor.b    #0x01,&P1OUT          ; toggle P1.0
             mov.w    #0,r15                ; use P15 as delay counter

delayloop:   dec.w    r15                   ; delay over?
             jnz      delayloop             ; n
             jmp      mainloop              ; y, toggle led
```

4 cycles

1 cycle

1 cycle

2 cycles

2 cycles