Instruction Sets

COEN 421/6341: Embedded Systems Design

1

Outline

- Introduction
- Preliminaries
- ARM Processor
- PICmicro Midrange Family
- TI C55x DSP
- TI C64x

COEN 421/6341: Embedded Systems Design

2

Introduction

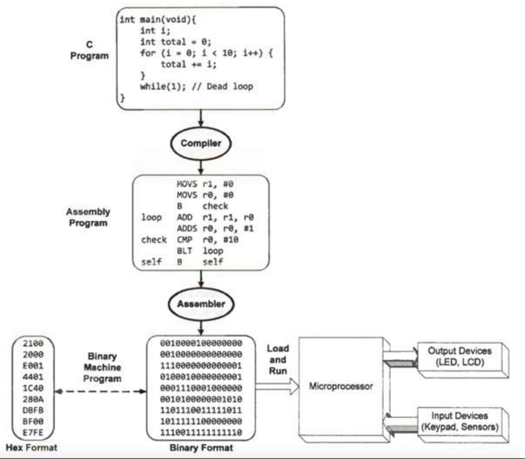
- Instruction Sets: the programmer’s interface to the hardware
- Use as much as possible in high-level languages
- Why should we look at instruction sets?
  - It is the key to analyzing the performance of programs
  - We gain insight into alternative ways to implement a particular function
- CPU as examples:
  - ARM
  - PIC16F
  - Texas Instruments C55x and C64x

Preliminaries

- A Review of Computer Architecture Taxonomy
  - CPU
    - Registers
    - Program counter (PC)
    - The CPU fetches the instruction from memory, decodes the instruction, and executes it
  - Memory
    - Holds both data and instructions
    - Can be read or written when given an address

Preliminaries

- Translate a C program into a machine program



```c
int main(void){
    int i;
    int total = 0;
    for (i = 0; i < 10; i++) {
        total += i;
    }
    while(1); // Dead loop
}
```

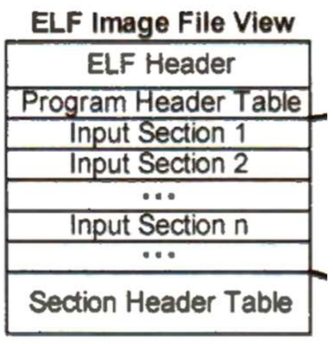
```
        MOVS r1, #0
        MOVS r0, #0
        B    check
loop    ADD  r1, r1, r0
        ADDS r0, r0, #1
check   CMP  r0, #10
        BLT  loop
self    B    self
```

```
2100    0010000100000000
2000    0010000000000000
E001    1110000000000001
4401    0100010000000001
1C40    0001110001000000
280A    0010100000001010
DBFB    1101110011111011
BF00    1011111100000000
E7FE    1110011111111110
```

5

Preliminaries

- Executable files created by compliers are usually platform dependent
- The organization of a machine program must follow a standard
- ELF (executable and linkable format) is a standard for binary machine program



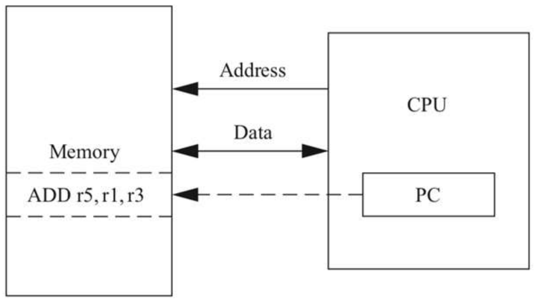
6

Preliminaries

- Executable Program:
  - Text segment:
    - binary machine instructions
  - Read-only data segment:
    - Variables unalterable at runtime
  - Read-write data segment:
    - Sets the initial value of statically allocated and modifiable variables
  - Zero-initialized data segment:
    - Holds all uninitialized variables declared in the program

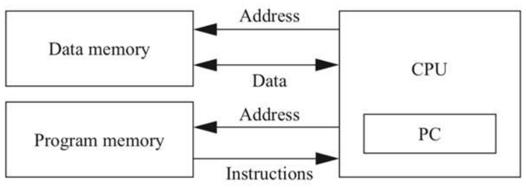
Preliminaries

- Von Neumann architecture
  - A computer whose memory holds both data and instructions
    - All sections of an executable program are loaded into the main memory
    - **Instruction fetch** and a **data operation** cannot occur at the same time
      - A single common bus is used for transfer
    - Accidental corruption of program memory may occur as data memory and program are stored physically in the same chip
    - Relatively inexpensive and simple



Preliminaries

- Harvard architecture
  - Separate memories for data and program
  - PC points to program memory, not data memory
  - Higher performance for digital signal processing
    - Processing of large amount of data and data streaming
    - When data must be processed at precise intervals



Preliminaries

- Complex instruction set computers (CISC)
  - Instructions that may perform very complex tasks
  - Single instruction can execute several low-level operations
  - Number of different instruction formats of varying lengths

- Reduced instruction set computers (RISC)
  - Fewer and simpler instructions
  - Use load/store instruction sets
    - Operations cannot be performed directly on memory locations, only on registers
  - Efficiently executed in pipelined processors

Preliminaries

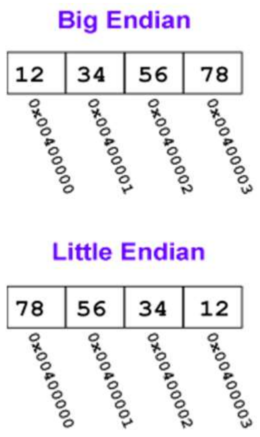
- IS Characteristics
  - Fixed versus variable length
  - Numbers of operands
  - Types of operations supported

Preliminaries

- Other possible architectures characterization
  - Word length:
    - It is more efficient for computers to load, process, store and transmit a group of bits simultaneously
    - Thus, bits are divided into a sequence of fixed-length logic units
    - Half-word: 16-bits
    - Word: 32-bits
    - Double-word: 64 bits

Preliminaries

- Other possible architectures characterization
  - The way they number bits, bytes, and words
    - Little-endian
      - Lowest-order byte residing in the low-order bits of the word
    - Big-endian
      - Lowest-order byte stored in the highest bits of the word



Preliminaries

- Processors can be characterized by:
  - Their instruction execution
    - Single-issue:
      - One instruction at a time
      - It may have several instructions at different stages of execution, only one can be at any particular stage of execution
    - Multiple-issue:
      - Multiple instructions every clock cycle

Preliminaries

- Processors can be characterized by:
  - Their instruction execution
    - Superscalar processor:
      - Uses specialized logic to identify at run time instructions that can be executed simultaneously
      - Use too much energy and are too expensive for widespread use in embedded systems
    - Very Long Instruction Word (VLIW):
      - Processor relies on the compiler to determine what combinations of instructions can be legally executed together

Preliminaries

- Registers: Fastest data storage in a computer system
- General-purpose registers:
  - Available for use by programs
- Special-purpose registers:
  - Used for internal operations and are unavailable to programmers

Preliminaries

- Assembly Languages
  - Usually share the same basic features
    - One instruction appears per line
    - Labels, which give names to memory locations, start in the first column
    - Instructions must start in the second column or after to distinguish them from labels
    - Comments run from some designated comment character to the end of the line

```
label1    ADR r4,c
          LDR r0,[r4]           ; a comment
          ADR r4,d
          LDR r1,[r4]
          SUB r0,r0,r1          ; another comment
```

17

Preliminaries

- CPUs can execute programs faster if they can execute more than one instruction at a time
- If the operands of one instruction depend on the results of a previous instruction, then the CPU cannot start the new instruction until the earlier instruction has finished
- Adjacent instructions may not directly depend on each other
  - CPU can execute several simultaneously
- How to parallelize instructions?

18

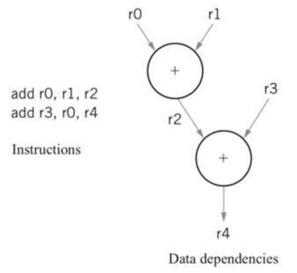
Preliminaries

- Opportunities for parallelism arise because many combinations of instructions do not introduce data or control dependencies
- Superscalar processors
    - A superscalar processor scans the program during execution to find sets of instructions that can be executed together
    - More expensive in both cost and energy consumption
    - Desktops and laptops
- VLIW processors:
    - Rely on the compiler to identify sets of instructions that can be executed in parallel
    - A set of instruction is bundled together into a **VLIW packet**
        - Contains a set of instructions that may be executed together
        - The execution of the next packet will not start until all the instructions in the current packet have finished executing
        - The compiler identifies packets by analyzing the program to determine sets of instructions that can always execute together

Preliminaries

Data dependency versus control dependency

- Data dependencies
    - It is a relationship between the data operated on by instructions

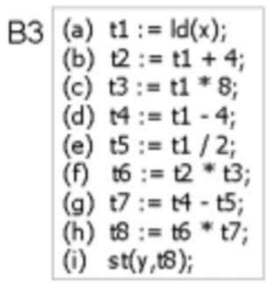


- Control dependencies
    - Instruction has a control dependency on a preceding instruction

```
if (a == b)
    a = a + b
b = a + b
```

Data Dependency

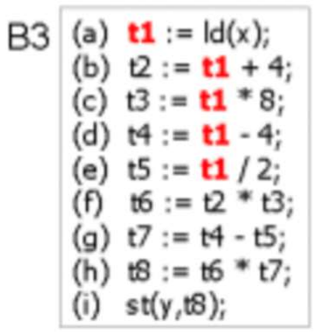
- Example



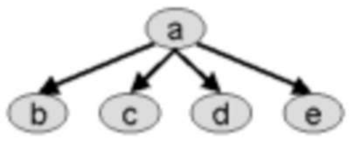
```
B3 (a) t1 := ld(x);
   (b) t2 := t1 + 4;
   (c) t3 := t1 * 8;
   (d) t4 := t1 - 4;
   (e) t5 := t1 / 2;
   (f) t6 := t2 * t3;
   (g) t7 := t4 - t5;
   (h) t8 := t6 * t7;
   (i) st(y,t8);
```

Data Dependency

- Example

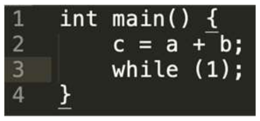


```
B3 (a) t1 := ld(x);
   (b) t2 := t1 + 4;
   (c) t3 := t1 * 8;
   (d) t4 := t1 - 4;
   (e) t5 := t1 / 2;
   (f) t6 := t2 * t3;
   (g) t7 := t4 - t5;
   (h) t8 := t6 * t7;
   (i) st(y,t8);
```

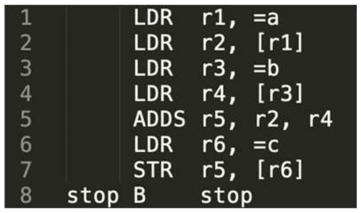


Data Dependency

- Example 2



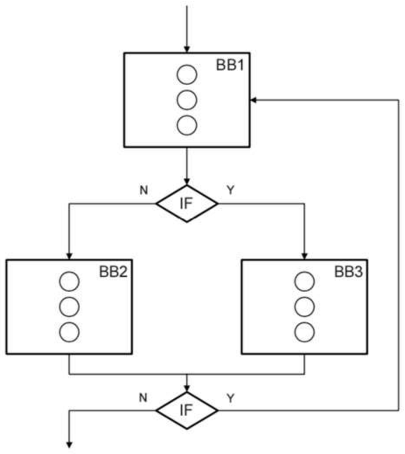
```
int main() {
    c = a + b;
    while (1);
}
```



```
        LDR  r1, =a
        LDR  r2, [r1]
        LDR  r3, =b
        LDR  r4, [r3]
        ADDS r5, r2, r4
        LDR  r6, =c
        STR  r5, [r6]
   stop B    stop
```

Preliminaries

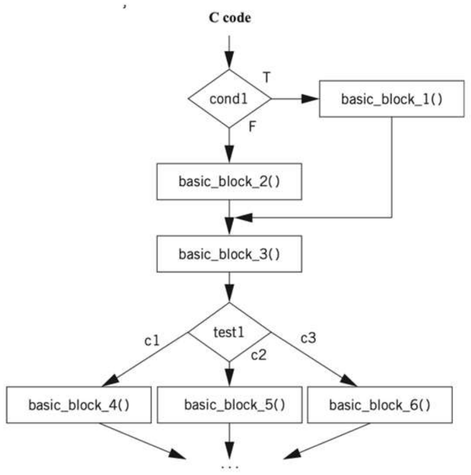
- Control-Data Flow Graph (CDFG)



Preliminaries

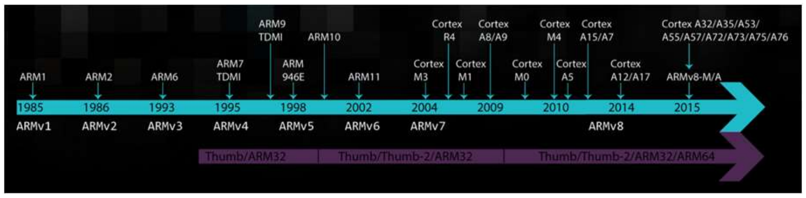
- Example:

```
if (cond1)
        basic_block_1();
else
        basic_block_2();
basic_block_3();
switch (test1) {
        case c1: basic_block_4(); break;
        case c2: basic_block_5(); break;
        case c3: basic_block_6(): break;
}
```



ARM Instruction Set

- ARM
  - A family of RISC architectures



https://azeria-labs.com/downloads/Slides-SAS_final.pdf

  - Do not manufacture its own chip
    - It licenses its architecture to companies
    - Companies either manufacture the CPU itself or integrate the ARM processor into a larger system

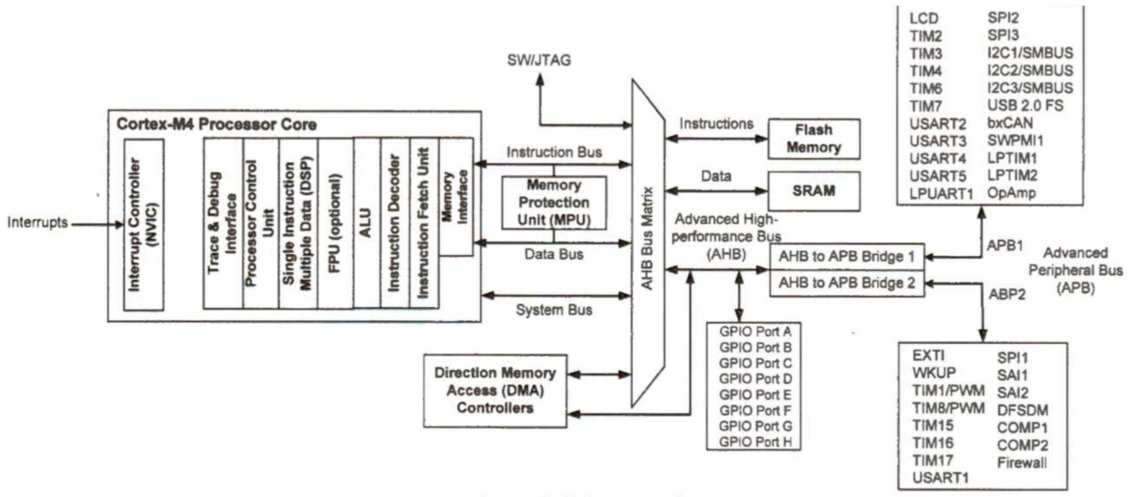
ARM Instruction Set

- Cortex: Prominent ARM family
  - Cortex-M series:
    - Excellent tradeoff between performance, cost, and energy effiency
    - Suitable for a broad range of microcontroller applications (e.g., IoT)
  - Cortex-A series:
    - Fast performance for sophisticated devices (e.g., smartphones and tablets)
  - Cortex-R
    - Designed for mission-critical real-time systems

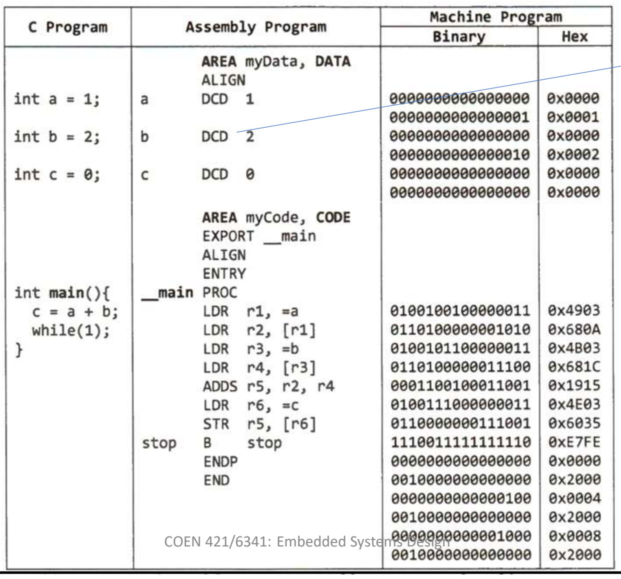
ARM Instruction Set

- Supported assembly instruction sets:
  - Thumb:
    - 16-bits in length instructions
    - Limits the number of registers that are accessible by an instructions
  - ARM32:
    - 32-bits in length instructions
    - More operand options
    - Larger immediate numbers
    - More addressable registers
  - Thumb-2:
    - 16-bits Thumb instructions + subset of 32-bit ARM32 instructions
  - ARM64:
    - 64-bit assembly instructions

ARM Cortex-M Organization



Executing a Machine Program



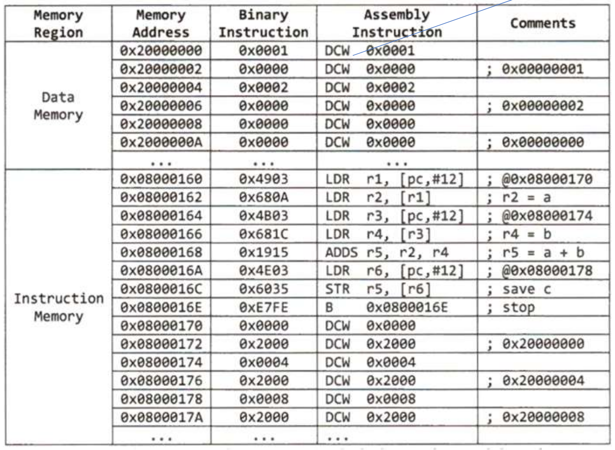
C Program	Assembly Program	Machine Program	
		Binary	Hex
	AREA myData, DATA		
	ALIGN		
int a = 1;	a DCD 1	0000000000000000	0x0000
		0000000000000001	0x0001
int b = 2;	b DCD 2	0000000000000000	0x0000
		0000000000000010	0x0002
int c = 0;	c DCD 0	0000000000000000	0x0000
		0000000000000000	0x0000
	AREA myCode, CODE		
	EXPORT __main		
	ALIGN		
	ENTRY		
int main(){	__main PROC		
c = a + b;	LDR r1, =a	0100100100000011	0x4903
while(1);	LDR r2, [r1]	0110100000001010	0x680A
}	LDR r3, =b	0100101100000011	0x4B03
	LDR r4, [r3]	0110100000011100	0x681C
	ADDS r5, r2, r4	0001100100011001	0x1915
	LDR r6, =c	0100111000000011	0x4E03
	STR r5, [r6]	0110000000111001	0x6035
	stop B stop	1110011111111110	0xE7FE
	ENDP	0000000000000000	0x0000
	END	0010000000000000	0x2000
		0000000000000100	0x0004
		0010000000000000	0x2000
		0000000000001000	0x0008
		0010000000000000	0x2000

Allocate one or more words of data

Executing a Machine Program

- Loading a Program

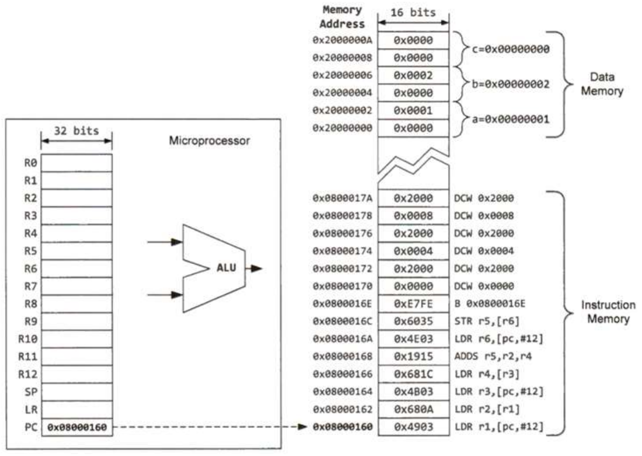
Allocate one or more halfword



Memory Region	Memory Address	Binary Instruction	Assembly Instruction	Comments
Data Memory	0x20000000	0x0001	DCW 0x0001	
	0x20000002	0x0000	DCW 0x0000	; 0x00000001
	0x20000004	0x0002	DCW 0x0002	
	0x20000006	0x0000	DCW 0x0000	; 0x00000002
	0x20000008	0x0000	DCW 0x0000	
	0x2000000A	0x0000	DCW 0x0000	; 0x00000000
	...	...	...	
Instruction Memory	0x08000160	0x4903	LDR r1, [pc,#12]	; @0x08000170
	0x08000162	0x680A	LDR r2, [r1]	; r2 = a
	0x08000164	0x4B03	LDR r3, [pc,#12]	; @0x08000174
	0x08000166	0x681C	LDR r4, [r3]	; r4 = b
	0x08000168	0x1915	ADDS r5, r2, r4	; r5 = a + b
	0x0800016A	0x4E03	LDR r6, [pc,#12]	; @0x08000178
	0x0800016C	0x6035	STR r5, [r6]	; save c
	0x0800016E	0xE7FE	B 0x0800016E	; stop
	0x08000170	0x0000	DCW 0x0000	
	0x08000172	0x2000	DCW 0x2000	; 0x20000000
	0x08000174	0x0004	DCW 0x0004	
	0x08000176	0x2000	DCW 0x2000	; 0x20000004
	0x08000178	0x0008	DCW 0x0008	
	0x0800017A	0x2000	DCW 0x2000	; 0x20000008
	...	...	...	

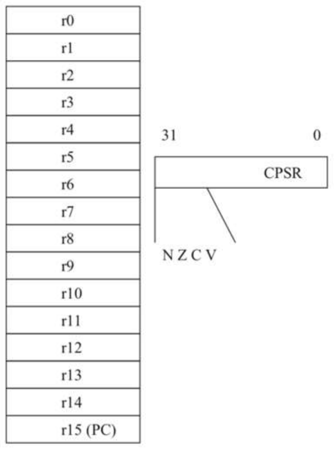
Executing a Machine Program

- Starting the execution


Memory Address
16 bits
0x2000000A 0x0000
0x20000008 0x0000
0x20000006 0x0002
0x20000004 0x0000
0x20000002 0x0001
0x20000000 0x0000
c=0x00000000
b=0x00000002
a=0x00000001
Data Memory
32 bits
Microprocessor
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
SP
LR
PC 0x08000160
ALU
0x0800017A 0x2000 DCW 0x2000
0x08000178 0x0008 DCW 0x0008
0x08000176 0x2000 DCW 0x2000
0x08000174 0x0004 DCW 0x0004
0x08000172 0x2000 DCW 0x2000
0x08000170 0x0000 DCW 0x0000
0x0800016E 0xE7FE B 0x0800016E
0x0800016C 0x6035 STR r5,[r6]
0x0800016A 0x4E03 LDR r6,[pc,#12]
0x08000168 0x1915 ADDS r5,r2,r4
0x08000166 0x681C LDR r4,[r3]
0x08000164 0x4B03 LDR r3,[pc,#12]
0x08000162 0x680A LDR r2,[r1]
0x08000160 0x4903 LDR r1,[pc,#12]
Instruction Memory

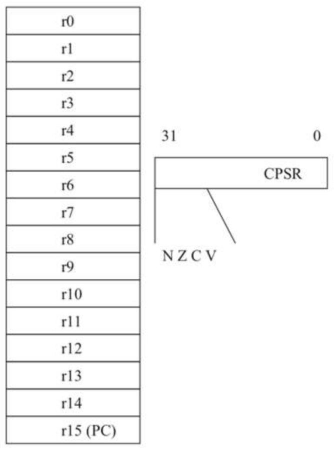
ARM Instruction Set

- ARM: Load-store architecture
  - Arithmetic and logical operations cannot be performed directly on memory locations
  - Data operands must first be loaded into the CPU and then stored back to main memory to save the results
- 16 general-purpose registers (r0 to r15)
  - **r0 to r14:** operation that can be done on one of them can be done on the others
  - **r15: PC**
    - The program counter should of course not be overwritten for use in data operations



ARM Instruction Set

- Current Program Status Register (CPSR)
  - It is set automatically during every arithmetic, logical, or shifting operation
  - Top four bits:
    - The negative (N) bit is set when the result is negative in two's-complement arithmetic
    - The zero (Z) bit is set when every bit of the result is zero
    - The carry (C) bit is set when there is a carry out of the operation
    - The overflow (V) bit is set when an arithmetic operation results in an overflow



Representation

- Carry and Overflow (CPSR register)
  - To understand how the CPSR flags are set, we need to understand the following:
    - Unsigned integers:
      - n bits can represent a total of 2^n symbols
      - Range: [0, 2^n – 1]
    - Signed integers: (representation)
      - Sign-and-magnitude
      - One's complement
      - Two's complement

Sign-and-Magnitude

- Uses the most significant bit to represent the sign
- The rest of bits are used to represent the magnitude
- E.g.: n=3
  - 000 = + 0
  - 001 = 1
  - 010 = 2
  - 011 = 3
  - 100 = - 0
  - 101 = -1
  - 110 = -2
  - 111 = -3

Range: [-2^(n-1) + 1, 2^(n-1) – 1]
Unique numbers: 2^n - 1
Two Zeros

One’s Complement

- Negative number is represented by inverting every bit of its positive equivalent
- E.g.: n=3
  - 000 = + 0
  - 001 = 1
  - 010 = 2
  - 011 = 3
  - 100 = -3
  - 101 = -2
  - 110 = -1
  - 111 = -0

Range: [-2^(n-1) + 1, 2^(n-1) – 1]
Unique numbers: 2^n - 1
Two Zeros

Two’s Complement

- Invert every bit of its positive equivalent (one’s complement)
- Add 1 to the one’s complement
- E.g.: n=3
  - 000 = 0
  - 001 = 1
  - 010 = 2
  - 011 = 3
  - 100 = -4
  - 101 = -3
  - 110 = -2
  - 111 = -1

Range: [-2^(n-1), 2^(n-1) – 1]
Unique numbers: 2^n
One Zero

Carry Flag

- Unsigned addition
  - Addition of two unsigned integers:
    - Carry if the result is larger than the maximum representable unsigned integer
  - E.g.: n=5 bits
  - Addition: 28+6. Will the carry flag be set?

Carry/Borrow Flag

- Unsigned addition
  - Addition of two unsigned integers:
    - Carry if the result is larger than the maximum representable unsigned integer
  - E.g.: n=5 bits
  - Addition: 28+6=?. Will the carry/borrow flag be set?

Carry/Borrow Flag

- Unsigned addition
  - Subtraction of two unsigned integers:
    - Borrow flag is set if the result is negative, i.e., smaller than the smallest expressible unsigned integer
  - E.g.: n=5 bits
  - Addition: 3-5= -2? 30?. Will the carry/borrow flag be set?
  - On ARM Cortex-M processors, the carry flag and the borrow flag are physically the same flag bit in the APSR
    - SUBTRACTION: Carry flag = 0 -> borrow has occurred on unsigned subtraction

Carry/Borrow Flag

- Summary: (unsigned addition/subtraction)
  - Carry flag is set if the sum is too large to be stored in the considered n-bit register
  - Carry flag is set if no borrow occurs (difference is positive or zero). Othersie, the carry bit is cleared.

Overflow Flag

- Signed numbers: (Overflow flag is set if):
  - Adding two positive numbers produce a non-positive number
  - Adding two negative numbers result a non-negative number

  - E.g. n=5
  - 12+5= 17? -15?

  - Overflow flag is set!

Overflow Flag

- Signed numbers: (Overflow flag is set if):
  - Adding two positive numbers produce a non-positive number
  - Adding two negative numbers result a non-negative number

  - E.g. n=5
  - 13-7= 20? 12? (p.s. extra bit is discarded)

  - Overflow flag is set!

Overflow Flag

- Signed numbers: (Overflow flag is set if):
  - Subtracting a positive number from a negative one creates a positive result
  - Subtracting a negative number from a positive one creates a negative result

  - Detecting overflow on subtraction can be converted to detecting overflow of addition

  A-B = A + (-B)
  A-B = A + TC(B)

  - E.g. n=5
  -9 - (+6) = ?

  - Overflow flag is set? No, since the addition of two negative numbers did not result in a non-negative number

Carry/Overflow Flag

- Assume:
  a = 0b10000
  b = 0b10000

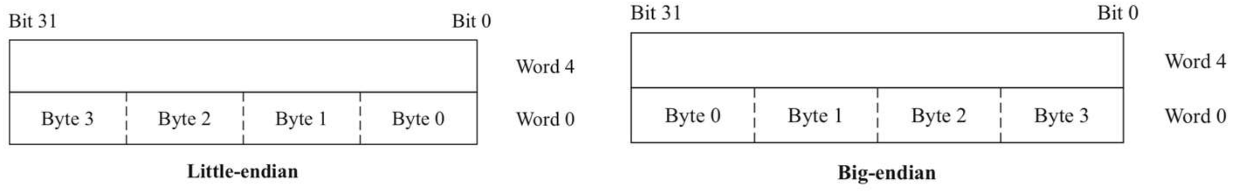
- What is the result of carry/overflow flag when a+b?
  - If unsigned integers carry flag will be set
  - If signed integers overflow flag will be set

Carry/Overflow Flag

- Assume:
  a = 0b10000
  b = 0b10000
- When adding a and b, the processor does not know whether a and b are signed or unsigned integers.
- It simply set both flags in this case
- It is the assembly programmer's responsibility to interpret the flag result

ARM Instruction Set

- ARM instructions are 32-bit long
- ARM7 allows addresses to be 32 bits long
  - Memory is a linear array of bytes addressed from 0 to 2^{32}-1
  - An address refers to a byte, not a word
  - Address the bytes in a word in either little-endian or big-endian mode



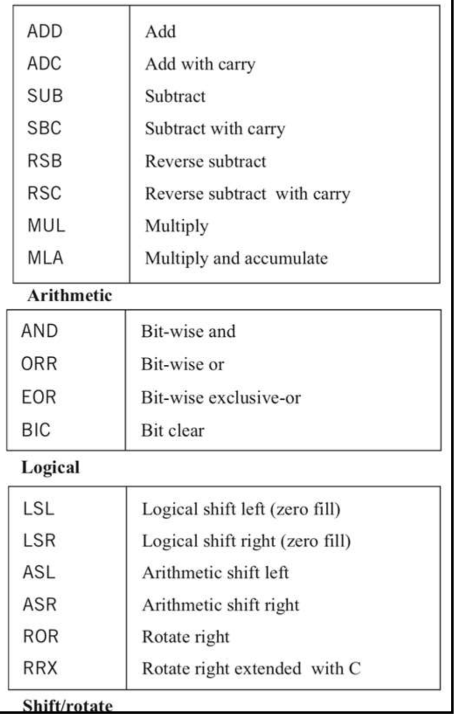
ARM Instruction Set

- Data Operations
  - Arithmetic
  - Logical
  - Shift/rotate
- Flow Control

49

ARM Instruction Set

- Basic form of a data instruction
  - ADD r0, r1, r2
- Instructions may also provide immediate operands
  - ADD r0, r1, #2



ADD	Add
ADC	Add with carry
SUB	Subtract
SBC	Subtract with carry
RSB	Reverse subtract
RSC	Reverse subtract with carry
MUL	Multiply
MLA	Multiply and accumulate

Arithmetic

AND	Bit-wise and
ORR	Bit-wise or
EOR	Bit-wise exclusive-or
BIC	Bit clear

Logical

LSL	Logical shift left (zero fill)
LSR	Logical shift right (zero fill)
ASL	Arithmetic shift left
ASR	Arithmetic shift right
ROR	Rotate right
RRX	Rotate right extended with C

Shift/rotate

50

ARM Instruction Set

- Comparison instructions
  - They do not modify general-purpose registers
  - They set the values of the NZCV bits of the CPSR register

CMP	Compare
CMN	Negated compare
TST	Bit-wise test
TEQ	Bit-wise negated test

- CMP r0, r1
  - Computers r0-r1
  - Sets the status bits
  - Throws away the result of the subtraction

ARM Instruction Set

- Move instructions

- MOV r0, r1
  - Sets the value of r0 to the current value of r1
- MVN
  - Complements the operand bits (one's complement) during the move

MOV	Move
MVN	Move negated

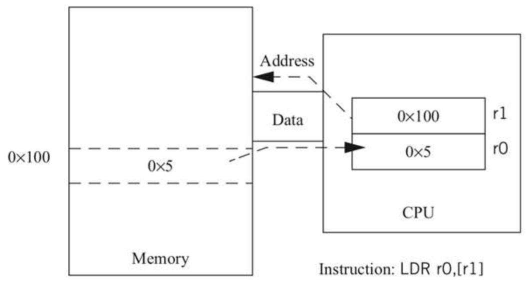
ARM Instruction Set

- Load/Store instructions
  - Used to transfer values between registers and memory
- LDRB and STRB load and store bytes rather than whole words
- LDRH and SDRH operate on half-words

LDR	Load
STR	Store
LDRH	Load half-word
STRH	Store half-word
LDRSH	Load half-word signed
LDRB	Load byte
STRB	Store byte
ADR	Set register to address

ARM Instruction Set

- An ARM address may be 32 bits long
- The ARM load and store instructions do not directly refer to main memory addresses
  - 32-bit address would not fit into an instruction that included an opcode and operands
- ARM uses **register-indirect addressing**
  - The value stored in the register is used as the address to be fetched from memory
  - E.g., LDR r0, [r1]



ARM Instruction Set

- E.g.: x = (a + b) – c   and y=a*(b+c)
  - r0 for a, r1 for b, r2 for c, and r3 for x
  - Must load the values of a, b and c to the registers
  - Must store the value of x back to memory

```
ldr r2, [fp, #-24]
ldr r3, [fp, #-28]
add r2, r2, r3
ldr r3, [fp, #-32]
rsb r3, r3, r2
str r3, [fp, #-36]
```

```
ldr r2, [fp, #-28]
ldr r3, [fp, #-32]
add r2, r2, r3
ldr r3, [fp, #-24]
mul r3, r2, r3
str r3, [fp, #-40]
```

ARM Instruction Set

- Addressing mode:
  - Register, immediate and indirect
  - Base-plus-offset addressing
    - Rather than using a register value directly as an address, the register value is added to another value to form the address
    - E.g., LDR r0, [r1, #16]
      - r1 is the base
      - The immediate value is the **offset**
    - **Autoindexing and postindexing**
      - Autoindexing: update the base register (e.g., LDR r0, [r1, #16]!)
      - Postindexing: does not perform the offset calculation until after the fetch has been performed (e.g., LDR r0, [r1], #16)
        - Will load the value stored in r1 to r0 and after will update r1 by adding 16

ARM Instruction Set

- Flow of Control
  - Branch (**B**) Instruction to change the flow of control
    - PC-relative, i.e., the branch specifies the offset from the current PC value to the branch target
    - We often wish to branch conditionally
    - The ARM allows any instruction, including branches, to be executed conditionally

EQ	Equals zero	$Z = 1$
NE	Not equal to zero	$Z = 0$
CS	Carry set	$C = 1$
CC	Carry clear	$C = 0$
MI	Minus	$N = 1$
PL	Nonnegative (plus)	$N = 0$
VS	Overflow	$V = 1$
VC	No overflow	$V = 0$
HI	Unsigned higher	$C = 1$ and $Z = 0$
LS	Unsigned lower or same	$C = 0$ or $Z = 1$
GE	Signed greater than or equal	$N = V$
LT	Signed less than	$N \neq V$
GT	Signed greater than	$Z = 0$ and $N = V$
LE	Signed less than or equal	$Z = 1$ or $N \neq V$

ARM Instruction Set

- Flow of Control
  - Example:

```
if (a > b) {
      x = 5;
      y = c + d;
} else {
      x = c - d;
}
```

```
        ldr r2, [fp, #-24]
        ldr r3, [fp, #-28]
        cmp r2, r3
        ble .L2
.L2:    mov r3, #3
        str r3, [fp, #-40]
        ldr r2, [fp, #-32]
        ldr r3, [fp, #-36]
        add r3, r2, r3
        str r3, [fp, #-44]
        b .L3
        ldr r3, [fp, #-32]
        ldr r2, [fp, #-36]
        rsb r3, r2, r3
        str r3, [fp, #-40]
.L3
```

Code for the true block

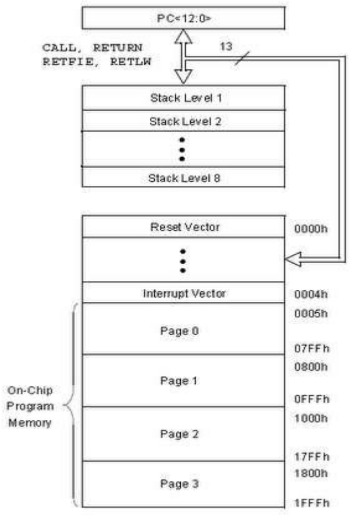
Code for the false block

PIC16F

- 8-bit word size and 14-bit instructions
- Harvard architecture with separate data and program memories
- Data memory is byte-addressable
- Provide several low power features
  - A sleep mode
  - The ability to select different clock oscillators to run at different speeds

PIC16F

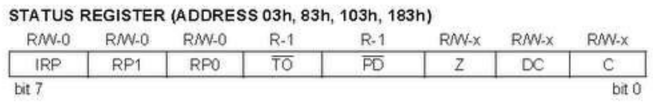
- Program Memory Organization
  - The PIC16F family uses a 13-bit program counter
  - Program Counter (PC) keeps track of the program execution by holding the address of the current instruction
  - 8-level deep x 13-bit wide hardware stack
    - Stack space is not part of either program or data space and the stack pointer is not readable or writable
  - The lowest addresses in memory hold the interrupt vectors
  - The rest of memory is divided into four pages
  - The low-end devices have access only to page 0; the medium-range devices have access only to pages 1 and 2; high- end devices have access to all four pages



Instruction space for the PIC16F.

PIC16F

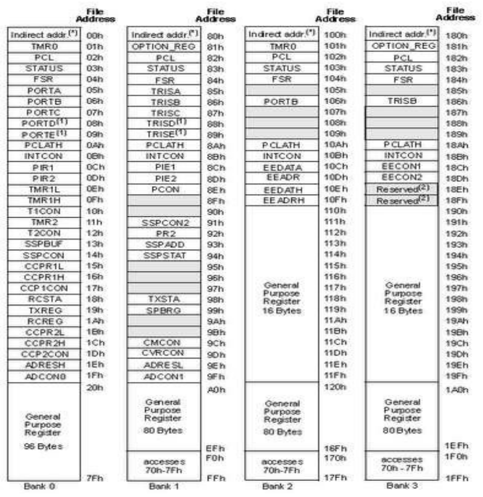
- Data Memory Organization
  - Data memory is partitioned into multiple banks
  - Two bits of the STATUS register, RP<1:0>, select which bank is used



STATUS REGISTER (ADDRESS 03h, 83h, 103h, 183h)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

  - General Purpose Registers
    - Data memory location
  - Special Function Registers
    - Perform many different operations, primarily for the I/O devices



	File Address		File Address		File Address		File Address
Indirect addr.(*)	00h	Indirect addr.(*)	80h	Indirect addr.(*)	100h	Indirect addr.(*)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h		105h		185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h		107h		187h
PORTD(1)	08h	TRISD(1)	88h		108h		188h
PORTE(1)	09h	TRISE(1)	89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDATA	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved(2)	18Eh
TMR1H	0Fh		8Fh	EEADRH	10Fh	Reserved(2)	18Fh
T1CON	10h		90h	General Purpose Register 16 Bytes	110h	General Purpose Register 16 Bytes	190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h		95h		115h		195h
CCPR1H	16h		96h		116h		196h
CCP1CON	17h		97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah		9Ah		11Ah		19Ah
CCPR2L	1Bh		9Bh		11Bh		19Bh
CCPR2H	1Ch	CMCON	9Ch		11Ch		19Ch
CCP2CON	1Dh	CVRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Register 96 Bytes	20h	General Purpose Register 80 Bytes	A0h	General Purpose Register 80 Bytes	120h	General Purpose Register 80 Bytes	1A0h
			EFh		16Fh		1EFh
		accesses 70h-7Fh	F0h	accesses 70h-7Fh	170h	accesses 70h - 7Fh	1F0h
	7Fh		FFh		17Fh		1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

PIC16F

ADDLW	Add literal and W
BCF	Bit clear f
ADDWF	Add W and f
BSF	Bit set f
ANDLW	AND literal with W
ANDWF	AND W with f
COMF	Complement f
CLRF	Clear f
DECF	Decrement f
CLRW	Clear W
IORLW	Inclusive OR literal with W
INCF	Increment f
IORWF	Inclusive OR W with F
MOVF	Move f
MOVWF	Move W to f
MOVLW	Move literal to W
NOP	No operation
RLF	Rotate left F through carry
RRF	Rotate right F through carry
SUBWF	Subtract W from F
SWAPF	Swap nibbles in F
XORLW	Exclusive OR literal with W
CLRWDT	Clear watchdog timer
SUBLW	Subtract W from literal

BTFSC	Bit test f, skip if clear
BTFSS	Bit test f, skip if set
CALL	Call subroutine
DECFSZ	Decrement f, skip if 0
INCFSZ	Increment f, skip if 0
GOTO	Unconditional branch
RETFIE	Return from interrupt
RETLW	Return with literal in W
RETURN	Return from subroutine
SLEEP	GO into standby mode

TI C55x DSP

- It is an accumulator architecture
  - Many arithmetic operations are of the form:
    - accumulator = operand + accumulator
- Processor and memory organization
  - Register: mean any type of register in the programmer model
  - Accumulator: mean a register used primarily in the accumulator style
  - Word: 16-bit long
  - Longword: 32-bit long
  - Instructions: byte addressable
  - Some instructions operate on addressed bits in registers
  - Few to none of these registers are general- purpose registers like those of the ARM

TI C55x DSP

- Most registers are memory-mapped
  - Register has an address in the memory space
  - Can be referred to in assembly language in two different ways
    - Referring to its mnemonic name
    - Referring through its address
  - The program counter is **PC**
  - The program counter extension register **XPC** extends the range of the program counter
- The C55x has four 40-bit accumulator
  - Low-order bits 0-15 are referred to as AC0L, AC1L, AC2L, and AC3L
  - The high-order bits 16-31 are referred to as AC0H, AC1H, AC2H, and AC3H
  - Guard bits 32-39 are referred to as AC0G, AC1G, AC2G, and AC3G

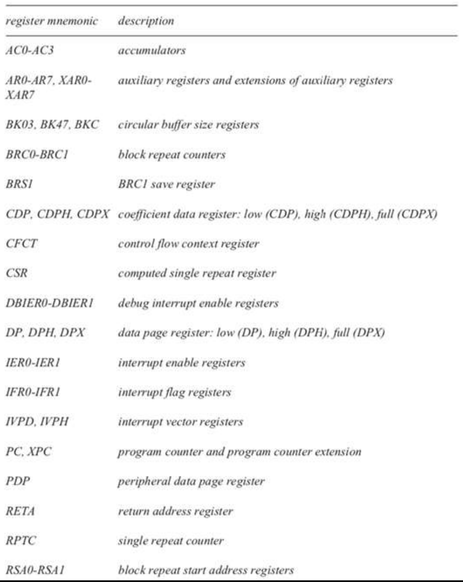
TI C55x DSP

- Six status registers
  - provide arithmetic and bit manipulation flags
  - a data page pointer and auxiliary register pointer
  - and processor mode bits, among other features
- The stack pointer SP keeps track of the system stack
- Eight auxiliary registers AR0-AR7 are used by several types of instructions, notably for circular buffer operations
- Several registers are used for block repeats - instructions that are executed several times in a row
- Four temporary registers, T0, T1, T2, and T3, are used for various calculations
- Two transition registers, TRN0 and TRN1, are used for compare-and-extract- extremum instructions
- Several registers are used for addressing modes
- Several registers control interrupts

65

TI C55x DSP

Registers in the TI C55x



register mnemonic	*description*
AC0-AC3	*accumulators*
AR0-AR7, XAR0-XAR7	*auxiliary registers and extensions of auxiliary registers*
BK03, BK47, BKC	*circular buffer size registers*
BRC0-BRC1	*block repeat counters*
BRS1	*BRC1 save register*
CDP, CDPH, CDPX	*coefficient data register: low (CDP), high (CDPH), full (CDPX)*
CFCT	*control flow context register*
CSR	*computed single repeat register*
DBIER0-DBIER1	*debug interrupt enable registers*
DP, DPH, DPX	*data page register: low (DP), high (DPH), full (DPX)*
IER0-IER1	*interrupt enable registers*
IFR0-IFR1	*interrupt flag registers*
IVPD, IVPH	*interrupt vector registers*
PC, XPC	*program counter and program counter extension*
PDP	*peripheral data page register*
RETA	*return address register*
RPTC	*single repeat counter*
RSA0-RSA1	*block repeat start address registers*

66

TI C64x

- The Texas Instruments TMS320C64x is a high-performance VLIW DSP
- The CPU can execute up to eight instructions per cycle using eight general-purpose 32-bit registers and eight functional units.
- The CPU is a load/store architecture
- On-chip memory is organized as separate data and program memories
- The external memory interface (EMIF) manages connections to external memory

TI C64x

- Instructions are fetched in groups known as **fetch packets**
- Due to the small size of some instructions, a fetch packet may include up to 14 instructions
- The instructions in a fetch packet may be executed in varying combinations of sequential and parallel execution
- An execute packet is a set of instructions that execute together
- Up to eight instructions may execute together in a fetch packet
  - But all must use a different functional unit
    - i.e., either performing different operations on a data path or using corresponding function units in different data paths

References

- Textbook – Chapter 2
- http://www.microcontrollerboard.com/pic_memory_organization.html#3Reg
- https://azeria-labs.com/downloads/Slides-SAS_final.pdf
- https://iitd-plos.github.io/col718/ref/arm-instructionset.pdf
- http://www.toves.org/books/arm/