ECE3849
D-Term 2021

Real Time Embedded Systems

Module 5 Part 4

Module 5 Part 4 Overview

- ARMv7 Instruction Set.
- Types of Instructions.
  - Data Processing.
  - Data Transfer.
  - Control Flow.
  - Special Purpose.

ARM Instruction Set: Getting Started

- The ARM Unified Assembler Language defines the syntax and operands for each instruction.
- The instruction set defines which sub-set of instructions are supported on a specific device.
  - 32-bit ARM Instructions.
  - 16-bit / 32-bit Thumb Instructions.
- TM4C1294 has a the following characteristics.
  - ARM Cortex-M4F processor.
  - ARMv7-M Architecture.
  - Thumb-2 Instruction set.
- When getting up to speed with an instruction set, it is good to keep it simple.
  - Use a programmer-friendly subset of the instruction set.
  - Focus on the most common instructions.
  - Have a reference handy.

Instruction categories

- Data processing
  - Arithmetic, logic, and shift operations.
  - Destination must be a CPU register.
  - Sources can be registers or immediate constants encoded in the instruction.
  - Can generate condition code bits (flags) if necessary for conditions statements like if, while, etc…
- Data transfer
  - Copying data from memory to registers or vice versa.
  - Manipulating the stack.
  - No arithmetic/logic/shift operations possible on the **data**.
  - Memory **address calculation** can use add, subtract and shift operations on register and immediate constants.
- Control flow
  - Branch (jump) to a different location in the code.
  - Call and return from a function.
  - Conditional branches are used to implement if(), while(), etc…
  - Writing to the PC (Program Counter) in a data processing or transfer instruction is another way to cause a branch.
- Special Purpose
  - Manipulating the PSR (Program Status Register).
  - Manipulating other special purpose registers.
  - Communicating with a coprocessor.

Common Data Processing

- CPU registers are referred to by "r" followed by their number.
  - CPU register 0 is r0, register 5 is r5.
- Three address instruction format
  - <opcode> Rd, Rn, <operand 2 / Rm>;
    - Opcode: the instruction to execute.
    - Rd: The destination / output of the instruction.
    - Rn: First operand / input.
    - <operand 2>: optional second operand / input.
  - Example:
    - add r2, r5, r0 ; implements r2 = r5 + r0.
    - add r2, r5, #125 ; implements r2 = r5 + 125.
- They have a two address instruction format
  - If Rd is the destination and an operand, then only two addresses are needed.
  - Example: r2 = r2 + r5;
  - Three address version: add r2, r2, r5
  - Two address version: add r2, r5
- Constants are preceded with #
  - #17 is a constant of value of decimal 17.
  - #0x11 is a the same constant in hexadecimal format.
  - 8-bit constants can be used immediately in the instruction.
  - Greater than 8-bits need to be loaded into a CPU register first.

Instruction Encoding

- Comments
  - The text after the semi-colon character is a comment.
- Why only 8-bits for constant?
  - Each instruction needs to be encoded into either a 16-bit or 32-bit instruction.
- add r2, r5, #125;   implements r2 = r5 +125

Encoding T3   Armv7-M

ADD{S}<c>.W <Rd>,<Rn>,#<const>

Rn = first operand = 5
4-bit CPU register number

Rd = destination = 2
4-bit CPU register number

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	Rn				0	imm3			Rd				imm8							

Specific opcode number

8-bit constant number = 125

- add r2, #125;   implements r2 = r2 + 125;

ADD<c> <Rdn>,#<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

Common Data Processing Instructions

Category	Operation	ALU opcode	Action
Arithmetic	**Add**	**ADD**	**Rd = Rn + <operand2>;**
	Add with carry	ADC	Rd = Rn + <operand2> + Carry;
	Subtract	**SUB**	**Rd = Rn - <operand2>;**
	Subtract with carry	SBC	Rd = Rn - <operand2> - !Carry;
	Reverse subtract	RSB	Rd = <operand2> - Rn;
Bitwise logical	**AND**	**AND**	**Rd = Rn & <operand2>;**
	Bit clear	BIC	Rd = Rn & ~<operand2>;
	OR	**ORR**	**Rd = Rn	<operand2>;**
	OR NOT	ORN	Rd = Rn	~<operand2>;
	Exclusive OR	**EOR**	**Rd = Rn ^ <operand2>;**

Shifter opcodes

Operation	Opcode	Action
Logical Shift Left	**LSL**	**<operand2> = Rm << <#shift>;**
Logical Shift Right	**LSR**	**<operand2> = (unsigned)Rm >> <#shift>;**
Arithmetic Shift Right	**ASR**	**<operand2> = (signed)Rm >> <#shift>;**

- LSR will shift in zeros: 1000_1100 >> 2 equals 0010_0011.
- ASR preserves the sign, it is a useful way to divide by powers of two.
  - If the MSB is 1 meaning it is negative value , ASR will shift in a one:
    1000_1100 >> 2 equals 1110_0011.
  - If the MSB is 0 meaning it is a positive value, ASR it will shift in a 0 just like the LSR function.

Data Processing: Convert C to Assembly

- Convert the following C code to assembly: a = b + 16 – (c & d)
- If the compiler is optimizing this code,
  - It knows which CPU registers are in use and will pick unused registers.
  - It will try to minimize the number of registers used at a time.
- For our code conversion,
  - We will assume all CPU registers are available and start with r0. Assume the values are already loaded into the register for this example.
  - We will try to minimize the registers in use by not creating unnecessary local variables.
  - First we need to assign a register to each variable we are using, then break the computation down into pieces. We will need one "and" operation, one subtract operation and one add operation.

ADD SUB AND

a = b + 16 – (c & d);

r0 r1 r2 r3

This uses r4 and r5 for local variables

```
and r4, r2, r3;
sub r5, r1, r4;
add r0, r5, #16;
```

r4 = c & d; → r4

r5 = b – (c &d)

a = $\underbrace{\text{b – (c\& d)}}_{\text{r5}}$ +16

This uses fewer registers

```
and r0, r2, r3;
sub r0, r1, r0;
add r0, r0, #16;
```

a = c & d; → a

a = b – (c &d)

a = $\underbrace{\text{b – (c \& d)}}_{\text{a}}$ +16

Data Transfer: Move

- Move instructions,
  - Move data between two registers.
  - Sets the register value to a constant.
- Two address instruction format.
  - <opcode> Rd, <operand 2 >
    - Opcode: the instruction to execute.
    - Rd: The destination / output of the instruction.
    - <operand 2>: The source / input of the instruction.
- Examples
  - mov r3, r8 ; r3 = r8 copies contents.
  - mov r3, #0x23fa ; r3 = 0x23fa sets all of r3 to the constant value.
    - Constants can be up to 16-bits for move operations.
  - movt r3, #0xffff ; r3[31:16] = 0xffff set only the top bits, leaves bottom bits unchanged.

  The result of the mov and movt commands combined is r3 = 0xffff23fa.

Category	Operation	Opcode	Action
Data movement	**Copy ("Move")**	`MOV`	`Rd = <operand2>;`
	Move into top	`MOVT`	`Rd[31:16] = <16-bit immediate>;`
Bitwise logical	**NOT**	`MVN`	`Rd = ~<operand2>;`

Data Transfer: Load Command

- Load command, ldr
  - Given a memory address, the load command copies the value from that memory location into the CPU registers.
  - The notation [r2] indicates an address is stored in r2.
  - If you consider your memory space as a huge array of bytes, [r2] is an index into that memory array, mem_{32}[r2];
- Load Examples
  - ldr r7, [r2] ; r7 = mem_{32}[r2]
    - This copies the data at address r2 into CPU register r7.
  - ldr r7, [r2, #12] ; r7 = mem_{32}[r2 +12]
    - If r2 = &a[0] (int array), then r7 = a[3] because an int is 4-bytes and 12 is the offset in bytes.
  - ldr r7, [r2,r9] ; r7 = mem_{32}[r2 +r9]

Data Transfer: Store Command

- Store command, str
  - Given a memory address, the store command copies a CPU register value into that memory location.
- Store Example
  - str r7, [r2] ; $mem_{32}[r2] = r7$
  - Just like the ldr command, offsets can be added to the address, [r2, #12] or [r2,r9]
- C conversion example (actual code is compiler specific)

```
int *p;    //p is a pointer to an integer.
int x;     //x is a 32-bit signed integer.
*p += x;  // *p = *p+x;  the value of p is incremented by x.
```

```
*p = += x
```

r0 holds the address of p

r1 holds the value x

ldr r2, [r0];	fetch the value of at $mem_{32}[r0]$
add r2, r1;	add the value p = p + x
str r2, [r0];	store the value r2 to $mem_{32}[r0]$

Data Transfer: Example

- temp and i are integers and already loaded in the CPU registers.
- a is an array of integers stored in memory
- This example swaps the values of a[i] and a[i+1]

Example

```
temp = a[i];        r0 = &a[0]
a[i] = a[i+1];      r1 = i
a[i+1] = temp;      r2 = temp.
```

- Step #1: Get the address of a[i].
  - a is an array of integers, each element is 4 bytes.
  - The address offset of a[i] from a[0], is i*4.
  - Shifting left by 2 is equivalent to multiply by 4.
  - &a[i] = &a[0] + i<<2.
- Step #2: temp = a[i].
  - r2 holds the temp value.
  - r3 holds the address of a[i].
- Step #3: load the value of a[i+1].
  - &a[i+1] = &a[i] +4 since int is a 4 byte data type.
  - r4 will hold the a[i+1] value.
- Step #4: a[i] = a[i+1].
  - Store the value of a[i+1],r4, to the address of a[i], [r3].
- Step #5: a[i+1] = temp.
  - Store the value of temp,r2, to the address of a[i+1],[r3+4].

```
lsl    r3, r1, #2 ;      r3 = i<<2 (address offset)
add r3, r0, r3;          r3 = &a[i]
ldr  r2, [r3]    ;       temp = a[i]
ldr r4, [r3, #4] ;       r4 = a[i+1]
str r4, [r3]       ;     a[i] = a[i+1]
str r2, [r3,#4]  ;       a[i+1] = temp
```

Control Flow Instructions

- Compare instructions are used in conjunction with branch instructions to implement conditional branching.
  - These are used when implementing if-else, case, while, for statements.
- Compare instructions have a 2-operand instruction format.
  - <opcode> Rn, <operand 2 >;

Category	Operation	Opcode	Action
Arithmetic	**Compare (Subtract)**	**CMP**	**Rn - <operand2>; // update flags**
	Compare Negative	CMN	Rn + <operand2>; // update flags
Bitwise logical	**Test (AND)**	**TST**	**Rn & <operand2>; // update flags**
	Test Equivalence (XOR)	TEQ	Rn ^ <operand2>; // update flags

- They do not have a destination defined in the instruction.
- Instead, the comparison result updates the condition code of bits (flags) in the APSR register.
  - N = negative flag
  - Z = zero flag
  - C = Carry flag (unsigned overflow on add, last bit shifted out)
  - V = Signed overflow flag
- For example,
  - If using the CMP instruction and Rn - <operand2> was less than zero, then the N-flag in the APSR register would be set.
  - If using the TST instruction and Rn & <operand2> was zero, then the Z-flag in the ASPSR register would be set.

Control Flow: Branch

- The branch commands then use the APSR flags to determine if a condition has been met and jump to a new location.
  - The location to jump to is given a label.
- Branch instructions have the following format.
  - B{cond} <label>
    - {cond} is the condition being checked.
    - <label> is the text string of the label to jump to if the condition is met.
  - BEQ eqlabel
    - Jumps to eqlabel if the flags show the equal condition.
  - B no condition listed always jumps.
    - b label1

Condition Field {cond}	
Mnemonic	**Description**
EQ	Equal
NE	Not equal
CS / HS	Carry Set / Unsigned higher or same
CC / LO	Carry Clear / Unsigned lower
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always (normally omitted)

Control Flow: Example

- x and y are signed integers already in CPU registers
  - r0 = x
  - r1 = y

Example

```
If (x > 10) {
    x = y;
}
else {
    x++;
}
```

```
cmp r0, #10      ;   check x > 10 condition and set flags
ble  else1       ;   if x <= 10 then jump to else1 label
mov r0, r1       ;   x = y
b  endif1        ;   under all conditions jump to the end
else1
add r0, r0, #1   ;  x++
endif1
```

Putting it all together example

- Example:

```
int x, i;        //r0=x;  r1=i;
int A[10];    // r2 = &A[0];
x = A[0];
for (i = 1; i < 10; i++) {
    if (A[i] > x) {
        x = A[i];
    }
    A[i] = x;
}
```

```
; initialize x and i
ldr r0, [r2]       ;    x = A[0]
mov r1, #1         ;    i = 1 to start the for loop
; check to see if the for loop is done.
loop1              ;    label for start of loop
cmp r1, #10        ;    if i < 10 do the loop
bge done1          ;    if i >= 10 jump to done1, else do the loop
; Calculate the address of A[i] and load the value A[i]
lsl   r3, r1, #2   ;    r3 = i<<2 or i*4 (address offset of A[i])
add r3, r2, r3     ;    r3 = &A[i] (temporary)
ldr  r4, [r3]      ;    r4 =  A[i]   (temporary)
; Compare A[i] > x and branch
cmp r4, r0         ;    A[i] > x  same as A[i] – x > 0
ble   endif1       ;    if <= then jump to end of if, else continue
mov r0, r4         ;    x = A[i]
endif1
; update the A[i] value and increment the loop counter
str r0, [r3]       ; A[i] = x
add r1, r1, #1     ; i++ to increment loop index
b loop1            ; jump to beginning of loop
done1              ; label for end of loop
```

ALU and Shift in Single Instruction

- The ALU and shift functions can be combined in one instruction
  - <ALU opcode> Rd, Rn, Rm{, <shifter_opcode> #<immediate>}
  - <load/store opcode> Rd, [Rn, {-}Rm{, <shifter_opcode> #<immediate>}]

Example

A[i] += x >> 8

r0 = &A[0]

r1 = i

r2 = x

```
; first load A[i] into CPU register r3 (temporary)
        &A[0] +   i << 2

ldr r3, [r0, r1, lsl #2 ]    ; load A[i]
         A[i]       x >> 8   ; &A[i] = &A[0] + i << 2

add r3, r3, r2, lsr #8       ; A[i] = A[i] + x >> 8
str r3, [r0, r1, lsl #2 ]    ;  store A[i]
```

Setting Condition Codes Instructions with S-suffix

- Compare instructions are not the only instruction to update the APSR flags.
- Instructions that add an S suffix will also update the flags.
  - Example: sub**s** r2, r2, #1 ; r2 = r2—1
    - subs will perform the subtract and also update the flags based on the result.
- Arithmetic operations.
  - Updates the N, Z, C and V flags based on the result.
- Logical operations.
  - Updates the N and Z flags based on the result.
- Shifter with a non-arithmetic ALU opcode.
  - C = last bit shifted out.

Load / Store Suffixes

- Load instructions with suffixes
  - LDR : 32-bit instruction
    - Rd = Data
  - LDRB (B suffix): unsigned byte instruction
    - Rd[7:0] = Data, Rd[31:8] = 0
  - LDRSB (SB suffix): signed byte instruction
    - Rd[7:0] = Data, Rd[31:8] = Data[7]
  - LDRH (H suffix): unsigned half word instruction
    - Rd[15:0] = Data, Rd[31:16] = 0
  - LDRSH (SH suffix): signed half word instruction
    - Rd[15:0] = Data, Rd[31:16] = Data[15]
- Store instructions are similar, just storing instead of loading
  - STR, STRB, STRSB, STRH, STRSH
- Example
  - ldr**b** r3, [r0, #5]
    - r3[31:8] = 0,
    - r3[7:0] = data[7:0] stored at address [r0]+5

Load / Store Instruction Variations

- PC (Program Counter) – relative addressing
  - Example:
    - ldr r7, my_constant ;
    - Equivalent to: ldr r7, [PC, #my_constant] ;
  - Constant can be up to 12-bits.
- Auto-indexing with LDR / STR.
  - Pre-indexing.
    - ldr r3, [r1, #4]**!** **; pre-indexed**
    - Performs two operations first r1 = r1+4 followed by r3 = mem_{32}[r1].
  - Post-indexing.
    - ldr r3, [r1]**,** #4 **; post-indexed**
    - Performs two operations first r3 = mem_{32}[r1] followed by r1 = r1+4.

Loading and Storing Multiple Registers

- Load / Store multiple registers in one instruction (LDM / STM).
- IA Suffix: Increment After.
  - ldmia r2, {r5-r7, r0}
    - First sorts by index r0, r5, r6, r7 then loads data starting from [r2]. Order you type it in does not matter.
    - r0 = mem_{32}[r2]
    - r5 = mem_{32}[r2 + 4]
    - r6 = mem_{32}[r2 + 8]
    - r7 = mem_{32}[r2 + 12]
  - ldmia r6!, {r1, r2}
    - First sorts by index r1, r2. Second loads data starting from [r6] to r1 and r2. Updates the r6 address with the number of bytes transferred. In this case 4 bytes / load * 2 registers loaded.
    - r1 = mem_{32}[r6]
    - r2=mem_{32}[r6+4]
    - r6 = r6+ 8.
- DB Suffix: Decrement Before
  - stmdb r6!, {r1, r2}
    - First sorts by index r1, r2. Second decrements r6 by the number of bytes to be transferred. Third stores to memory.
    - r6 = r6 – 8
    - mem_{32}[r6] = r1
    - mem_{32}[r6+4] = r2
- See ece3849_arm_assembly.pdf for more examples.

Block Copy Example

Example:

```
unsigned long *src, *dst, count;
count &= ~0x7;        // round down to nearest multiple of 8
 while (count != 0) { // unroll this loop 8 times
     *dst = *src;
     dst++;      // pointer arithmetic
     src++;
     count--;
}
```

r0 = &src
r1 = &dec
r2 = count

```
; initialize count
bics r2, r2, #0x7       ; clear count  bit[2:0]
beq done1               ; if count == 0, your done
; else start the while loop
loop1   ldmia r0!, {r3-r10}  ; load 8 words starting from src address to CPU registers r3 to r10
        stmia r1!, {r3-r10}  ; store 8 words from r3 to r10 to address starting at dst address
        subs r2, r2, #8      ; update word count, count = count -8
        bne loop1             ; if remaining count != 0 then jump to loop1, else continue.
done1
```