CprE 288 – Introduction to Embedded Systems
Course Review for Exam 3

Instructors:
Dr. Phillip Jones
Announcements

• Final Exam:
  – Morning Section: Monday 12/12 (9:45am): 90 minutes
  – Afternoon Section: Tuesday 12/13 (Noon): 90 minutes
• Exam 3 location: Our regular classroom
• Allowed Textbook, 1 page of notes, and a Calculator
  – Notes must be a paper copy (not electronic)
Topics in Assembly Programming

• Data Movement
  – Move constants to registers
  – Move data between registers
  – Move data between register and memory (Load/Store)

• Logic & Arithmetic

• Control Flow
  – Test/compare register(s), set status register flags
  – Choose the right branch

• Function call convention
  – Pass parameters and return values
  – Share registers between Caller and Callee
  – The Stack and its usage

CprE 288, ISU, Fall 2011
Figure 2.5  A structure block diagram of 21 registers in the Cortex®-M4 Core.
• 16 32-bit general purpose registers
  – Used for accessing SRAM
  – Used for storing function parameters
  – Used for instructions to execute operations on

• What is an 32-bit register.
  – Basically just 32 D-Flips connected together

| Bit 31 | Bit 30 | Bit 5 | ... | ... | Bit 2 | Bit 1 | Bit 0 |
### Status Register (SREG)

#### (a) Three individual registers – APSR, IPSR, and EPSR.

<table>
<thead>
<tr>
<th>Bits</th>
<th>31 30 29 28 27 26:25 24 23:20 19:16 15:10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>APSR</td>
<td>N  Z  C  V  Q   GE*  Reserved</td>
</tr>
<tr>
<td>IPSR</td>
<td>Reserved  ICI/IT  T  Reserved  ICI/IT  Exception Number</td>
</tr>
<tr>
<td>EPSR</td>
<td>Reserved  ICI/IT  T  Reserved  ICI/IT  Reserved</td>
</tr>
</tbody>
</table>

#### (b) The combined register PSR.

<table>
<thead>
<tr>
<th>Bits</th>
<th>31 30 29 28 27 26:25 24 23:20 19:16 15:10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSR</td>
<td>N  Z  C  V  Q   ICI/IT  T     GE*   ICI/IT  Exception Number</td>
</tr>
</tbody>
</table>

Figure 2.6  Structure and bit functions in special registers.
Status Register (SREG): Common flags

- **Z: Zero flag**
  - Set to 1 when the result of an instruction is 0
- **C: Carry flag**
  - Set to 1 when the result of an instruction causes a carry to occur
- **N: Negative**
  - Set to 1 to indicate the result of an instruction is negative
- **V: Overflow**
  - Set to 1 to indicate the result of an instruction caused an overflow
Instructions to move data: Brief Summary

• **Move**
  - MOV Rd, Rt  
    Move Between Registers: \( Rd \leftarrow Rt \)
  - MOVW Rd, #Imm16  
    Constant to Register: \( Rd \leftarrow #Imm16 \)
  - MOVT Rd, #Imm16  
    Constant to upper Register: \( Rd \leftarrow #Imm16 \)

• **Load (Load to Register)**
  - LDR Rd, [Rn, #Offset]  
    Load 32-bit: \( Rd \leftarrow [Rn + #\text{Offset}] \)
  - LDRB Rd, [Rn, #Offset]  
    Load 8-bit: \( Rd \leftarrow [Rn + #\text{Offset}] \)
  - LDRH Rd, [Rn, #Offset]  
    Load 16-bit: \( Rd \leftarrow [Rn + #\text{Offset}] \)

• **Store (Store from Register)**
  - STR Rt, [Rn, #Offset]  
    Load 32-bit: \( [Rn + #\text{Offset}] \leftarrow Rt \)
  - STRB Rt, [Rn, #Offset]  
    Load 8-bit: \( [Rn + #\text{Offset}] \leftarrow Rt \)
  - STRH Rt, [Rn, #Offset]  
    Load 16-bit: \( [Rn + #\text{Offset}] \leftarrow Rt \)
  - STRD Rt, Rt2, [Rn, #Offset]  
    Load 64-bit: \( [Rn + #\text{Offset}] \leftarrow Rt \)
    \( [Rn + #\text{Offset}+4] \leftarrow Rt2 \)
Load/Store: Addressing modes

• Immediate offset
  – LDR Rt, [Rn, #K]  Regular Imm Offset \( \text{Rt} \leftarrow [\text{Rn} + K] \)
  – LDR Rt, [Rn, #K]!  Pre-Index Offset: \( \text{Rt} \leftarrow [\text{Rn} + K], \text{Rn} \leftarrow \text{Rn} + K \)
  – LDR Rt, [Rn], #K  Post-Index Offset: \( \text{Rt} \leftarrow [\text{Rn}], \text{Rn} \leftarrow \text{Rn} + K \)

• Register offset
  – LDR Rt, [Rn, Rm, LSL #n]  \( \text{Rt} \leftarrow [\text{Rn} + (\text{Rm} \ll n)] \)

• PC-Relative
  – LDR Rt, [PC, #K]  \( \text{Rt} \leftarrow [\text{PC} + K] \)

• PUSH/POP Addressing mode
  – Loads/Stores a list of registers to the stack

• Multiple Register Addressing mode
  – Loads/Stores a list of registers

• Exclusive Addressing mode
  – Used to guarantee a single source is accessing a memory
Brief overview of arithmetic instructions

**Addition**: ADD, ADC, ADDW

**Subtraction**: SUB, SBC

**Logic**: AND, ORR, EOR

**Shift**: LSL, LSR, ASR, ROR

**Compliments**: NEG

**Multiplication**: MUL, SMULL, UMULL
Given assembly instructions and their general encoding, provide their specific binary encoding.

For example, given the general encoding of \texttt{MOVt \ Rd, \ K}.
Where \( K \) is encoded as: \( K = \text{imm4} : i : \text{imm3} : \text{imm8} \)

What is the specific binary encoding for \texttt{MOVt \ R3, 0xFF33}
char ch1 = 0x30;
char ch2 = 0x40;
int a = 0x1010;

ch1 = ch2;

a = a + a;
Example Questions

```c
signed char ch1;
signed char ch2;
signed char flag;
int a;
int b;
signed char *pch;
int *pint;

*pch = ch1;
a = *pint;
pint = &b;
a = a * b;
ch1 = ch1 & ch2;
```
<table>
<thead>
<tr>
<th>Condition Encoding (cond)</th>
<th>Branch Type</th>
<th>Meaning</th>
<th>Status Flag State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>BEQ</td>
<td>Equal</td>
<td>Z = 1</td>
</tr>
<tr>
<td>0001</td>
<td>BNE</td>
<td>Not equal</td>
<td>Z = 0</td>
</tr>
<tr>
<td>0010</td>
<td>BHS</td>
<td>Higher or Same (Unsigned)</td>
<td>C = 1</td>
</tr>
<tr>
<td>0011</td>
<td>BLO</td>
<td>Lower (Unsigned)</td>
<td>C = 0</td>
</tr>
<tr>
<td>0100</td>
<td>BMI</td>
<td>Negative</td>
<td>N = 1</td>
</tr>
<tr>
<td>0101</td>
<td>BPL</td>
<td>Positive</td>
<td>N = 0</td>
</tr>
<tr>
<td>0110</td>
<td>BVS</td>
<td>Overflow</td>
<td>V = 1</td>
</tr>
<tr>
<td>0111</td>
<td>BCV</td>
<td>No overflow</td>
<td>V = 0</td>
</tr>
<tr>
<td>1000</td>
<td>BHI</td>
<td>Higher (Unsigned)</td>
<td>C = 1 &amp; Z = 0</td>
</tr>
<tr>
<td>1001</td>
<td>BLS</td>
<td>Lower or Same (Unsigned)</td>
<td>C = 0</td>
</tr>
<tr>
<td>1010</td>
<td>BGE</td>
<td>Greater than or Equal (Signed)</td>
<td>N = V</td>
</tr>
<tr>
<td>1011</td>
<td>BLT</td>
<td>Less than (Signed)</td>
<td>N != V</td>
</tr>
<tr>
<td>1100</td>
<td>BGT</td>
<td>Greater than (Signed)</td>
<td>N = V &amp; Z = 0</td>
</tr>
<tr>
<td>1101</td>
<td>BLE</td>
<td>Less than or Equal (Signed)</td>
<td>N != V</td>
</tr>
</tbody>
</table>
If-Statement: Structure

Control and Data Flow Graph

Linear Code Layout

- `cond`
- `if-body`
- `F`
- `T`
- `test cond`
- `br if cond=F`
- `if-body`
If-Else Statement: Structure

Control and Data Flow Graph

- Cond
  - F
    - If-body
    - Else-body
  - T
    - If-body
    - Else-body

Linear Code Layout

- Test cond
- Br if cond=F
  - If-body
    - Jump
  - Else-body
Example Questions

```c
int a, b, max;

if (a > b) {
    max = a;
} else {
    max = b;
}

What if

unsigned int a, b, max
```
int a;
signed char flag;

if (a <= 20)
   flag = 0;
else
   flag = 1;
Example Questions

```c
int a, b;
signed char flag;

if (flag)
    a = b;
else
    b = a;
```
If and If-Else summary

For simple If and If-Else statement, the behavior of C and Assembly are complementary. So complement condition is used.

Examples:

<table>
<thead>
<tr>
<th>C</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a &gt;= b)</td>
<td>branch if a&lt;b, use BLT</td>
</tr>
<tr>
<td>if (a &gt; b)</td>
<td>branch if a__b, use BLE</td>
</tr>
<tr>
<td>if(a = b)</td>
<td>branch if a!=b, use BNE</td>
</tr>
</tbody>
</table>
Caveat when comping against a constant

Translate the C code into an equivalent condition, then compile into assembly using complement condition.

**Case 1:** if \(5 > a\) \(\Leftrightarrow\) if \(a < 5\),

\[
\begin{align*}
\text{C} & \quad \text{translated C} \\
\text{Assembly} & \quad \text{branch if } a \geq 5 \\
& \quad \text{CMP R0, #5} \\
& \quad \text{BGE endif}
\end{align*}
\]
if (a >= b && b< 10 && ... && a > 20)

; If only consists of Boolean AND’s, follow complement rule

; assume a in R0, and b in R1
    CMP  R0,  R1
    BLT  else ; complement of >
    CMP  R1,  #10
    BGE  else ; complement of <
    ...
    CMP  R0,  #20
    BLE  else ; complement of >
    ...
    ; if-body
    B endif
else:
    ...
    ; else-body
endif:

Recall C uses Lazy Evaluation
Compound Condition (||)

if (a >= b || b < 10 || ... || a > 20)

;If only Boolean OR’s, then complement only last condition

; assume a in R0, and b in R1
  CMP  R0, R1
  BGE  if-body ; no complement, and br to if-body
  CMP  R1, #10
  BLT  if-body ; no complement, and br to if-body
      ...
  CMP  R0, #20
  BLE  else ; complement of >

if-body:
   ... ; if-body
   B  endif
else:
   ... ; else-body
endif:

Again, Recall C uses Lazy Evaluation
Function Call Convention

What is required for supporting a function call?

- Passing parameters
- Getting the return value
- Sharing registers between Caller and Callee
- Local storage (typically placed in reg or on the Stack)
- Jumping to the Callee
- Returning to the Caller

Why we study the C calling convention

- Must follow it when mixing C with assembly or using pre-compiled C library functions
- The calling convention is NOT part of the instruction set architecture. It is an agreed upon convention to allow a compiler and human to generate code that can work together
Function parameters

- Use R0, R1, R2, R3
- Parameters passed to R0 – R3 in order.
- Parameter size: 2, or 1 bytes
  - Value is Zero or Signed extended before placed in a register
- Parameter size 8 bytes
  - Use a pair of registers (e.g. R1:R0)
- Extra parameters placed on the Stack

Function return value

- 8-bit in R0 (Zero or Signed extended)
- 16-bit in R0 (Zero or Signed extended)
- 32-bit in R0
- 64-bit in R1:R0
How to share registers between Caller and Callee?

Non-volatile (Callee must preserve): R4-R11, SP

Callee must make sure these register values are preserved/restored. In other words, the Caller can assume the value of these registers after the Callee function returns will be the same as before the Callee function was called.

Volatile (Callee can freely modify): R0-R3, LR

If the Caller wants the values of these registers to be the same before and after the Callee function executes, then the Caller must preserve them before the Callee function executes, and restore them after Callee function executes.
### ARM Function Call Standard

#### Non-volatile: Callee must preserve/restore if it uses

#### Volatile: Caller must preserve/restore if it wants the register to maintain its value across a function call

- **R15**
- **R14**
- **R13**
- **R12**
- **R11**
- **R10**
- **R9**
- **R8**
- **R7**
- **R6**
- **R5**
- **R4**
- **R3 (a4)**
- **R2 (a3)**
- **R1 (a2)**
- **R0 (a1)**

#### Return value:
- 4 bytes or less
- 8 bytes

**a**: Function argument (Assuming each param is 4 bytes or less)
ARM: Typical Logical Memory Layout

- Static Data (e.g. Global vars)
- Stack (e.g. local vars, function args & return value, return address)
- Heap (e.g. dynamically allocated memory: malloc, new)
- Code (Code may be in the same or a separate memory device)

Stack Pointer (SP): Address of top of the stack

Stack Base Address
Stack grows downward

Heap grows upward

Code Segment

0x0000_0000

http://class.ece.iastate.edu/cpre288
Example Memory Layout

```c
static char[] greeting = "Hello world!";

main()
{
    int i;
    char bVal;

    LCD_init();
    LCD_PutString(greeting);
    ...
}
```

Memory Mapped I/O, Devices

Static Data

Heap (grows up)

Stack (grows down)

I/O addresses

Code

Low end Address

High end Address

http://class.ece.iastate.edu/cpre288
ARM: Typical Function Stack-Frame

- Stack is typically divided into a number of function stack-frames.
- Stack-Frame: Contains information associated with a function call
  - Function args, local vars, return address, preserved values

Stack Base Address
Stack grows downward

Function 1 Frame
- Function Return Address
- Preserved Values
- Function Args
- Local Vars

Function 2 Frame
- Function Return Address
- Preserved Values
- Function Args
- Local Vars

http://class.ece.iastate.edu/cpre288
Function and Stack

Function Frame: Local storage for a function

Figure 1 - Stack Frame creation and destruction
Function Call: Example

main: …

BL myfunc

…

myfunc: …

…

…

BX LR

; call setup
; call myfunc
; return to here

; prologue
; function body
; epilogue
; return
Making a Func call vs. Implementing a Func

Know the difference between how to make a function call, and how to implement a function.
How main() calls add3: Assume for some reason R3 and R5 must be preserved across the function call

// sum = add3(x, y, z);
// x @ 0x1000_0000, y @ 0x1000_0004, z @ 0x1000_0008
// sum @ 0x1000_000C

main:

    PUSH R3 ; Save R3
    MOVW R4, 0x0000 ; Get base address
    MOVT R4, 0x1000
    LDR R0, [R4, #0] ; load x ; 1st parameter of add3
    LDR R1, [R4, #4] ; load y ; 2nd parameter of add3
    LDR R2, [R4, #8] ; load z ; 3rd parameter of add3
    BL add3 ; call add3(x, y, z)
    STR R0, [R4, #C] ; store result to sum
    POP R3 ; Restore R3

Question: Is it not necessary to push/pop R5?
Example Question

How main() calls add3: Assume for some reason R3 and R5 must be preserved across the function call

```c
// sum = add3(x, y, z);
// x @ 0x1000_0000, y @ 0x1000_0004, z @ 0x1000_0008
// sum @ 0x1000_000C
main:
    PUSH R3 ; Save R3
    MOVW R4, 0x0000 ; Get base address
    MOVT R4, 0x1000
    LDR R0, [R4, #0] ; load x ; 1st parameter of add3
    LDR R1, [R4, #4] ; load y ; 2nd parameter of add3
    LDR R2, [R4, #8] ; load z ; 3rd parameter of add3
    BL add3 ; call add3(x,y, z)
    STR R0, [R4, #C] ; store result to sum
    POP R3 ; Restore R3
```

Question: Is it not necessary to push/pop R5?
DO-WHILE Loop

Control and Data Flow Graph

Linear Code Layout

Loop prologue (optional)

do-body

test cond

br if cond=T

Loop epilogue (optional)
WHILE Loop

Control and Data Flow Graph

Linear Code Layout

(optional prologue and epilogue not shown)
FOR Loop

Control and Data Flow Graph

- **init-expr**
  - **for-body**
  - **incr-expr**
  - **cond**
    - F
    - T

Linear Code Layout

- **init-expr**
  - **jump**
    - **for-body**
      - **Incr-expr**
      - **test cond**
        - **br if cond=T**

(oretical prologue and epilogue not shown)
// Copy the contents of array X into array Y.
// Both arrays have N elements.
void copyArray(int X[ ], int Y[ ], int N);
Example Questions

// Find out the maximum value of an array, return the value. The array has N elements
int maxOfArray(int X[ ], int N);