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

Instructors:
Dr. Phillip Jones
Announcements

• **Final Exam: 90 minutes**
  – **Exam 3 location:** Our regular classroom (See syllabus for day/time)
  – **You are allowed:** 1) Datasheet, 2) Textbook, 3) ARM Instruction Set Summary, 4) Calculator, 5) 1 sheet of notes (you can use both sides)
    • Notes must be a paper copy (or 10pt font or greater if electronic format)
  – Using any other resources is consider cheating an result in **receiving an F for CPRE 288**
Topics

General architecture

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
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

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 30</th>
<th>Bit 5</th>
<th>...</th>
<th>...</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
</table>

http://class.ece.iastate.edu/cpre288
Status Register (SREG)

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

(b) The combined register PSR.

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: $Rd \leftarrow [Rn + #Offset]$
  - LDRB Rd, [Rn, #Offset]  
  Load 8-bit: $Rd \leftarrow [Rn + #Offset]$
  - LDRH Rd, [Rn, #Offset]  
  Load 16-bit: $Rd \leftarrow [Rn + #Offset]$

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

- Immediate offset
  - LDR Rt, [Rn, #K]  
    Regular Imm Offset $Rt \leftarrow [Rn + K]$
  - LDR Rt, [Rn, #K]!
    Pre-Index Offset: $Rt \leftarrow [Rn + K]$, $Rn \leftarrow Rn+K$
  - LDR Rt, [Rn], #K
    Post-Index Offset: $Rt \leftarrow [Rn]$, $Rn \leftarrow Rn+K$

- Register offset
  - LDR Rt, [Rn, Rm, LSL #n]  
    $Rt \leftarrow [Rn + (Rm << n)]$

- PC-Relative
  - LDR Rt, [PC, #K]  
    $Rt \leftarrow [PC + K]$

- 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
Arithmetic Instruction

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: \( K = \text{imm4:i:imm3:imm8} \)

What is the specific binary encoding for \texttt{MOVT R3, 0xFFF33}
C to Assembly: Example Questions

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 *pchar;
int *pint;

*pchar = 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

T

F

if-body

test cond

br if cond=F

if-body
If-Else Statement: Structure

Control and Data Flow Graph

```
cond
  F
  T
  if-body
  else-body
```

Linear Code Layout

```
test cond
  br if cond=F
    If-body
    jump
    else-body
```
int a, b, max;

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

What if

unsigned int a, b, max
Example Questions

```c
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;
```
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.

C          translated C          Assembly
Case 1: if (5 > a) ⇔ if (a < 5),          branch if a ≥ 5
          CMP  R0, #5
          BGE   endif
if \((a \geq b \land b < 10 \land \ldots \land a > 20)\)

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

; assume \(a\) in \(R_0\), and \(b\) in \(R_1\)

```assembly
CMP    R0, R1
BLT    else ; complement of \(\geq\)
CMP    R1, #10
BGE    else ; complement of \(<\)
    ...
CMP    R0, #20
BLE    else ; complement of \(\geq\)
    ...
    ; 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
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

(jump
while-body
test cond
br if cond=T)

(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

(optional prologue and epilogue not shown)
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 a non-volatile register

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

**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)
static char[] greeting = "Hello world!";

main()
{
    int i;
    char bVal;

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

Memory Mapped I/O, Devices

High end Address

I/O addresses

Stack (grows down)

Heap (grows up)

Static Data

Code

Low end Address
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
Function and Stack

Function Frame: Local storage for a function
Function Call: Example

main: ...

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
  MOVW 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)
Example Question

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)
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

Linear Code Layout

(init-expr)

for-body

(increment-expr)

cond

T

F

(init-expr)

jump

(for-body)

(increment-expr)

(test cond)

(br if cond=T)

(optional prologue and epilogue not shown)
Example Questions

// Copy the contents of array X into array Y. // Both arrays have N elements.
void copyArray(int X[], int Y[], int N);
// Find out the maximum value of an array, return the value. The array has N elements
int maxOfArray(int X[], int N);