EECS 373 Midterm 1
Fall 2017

27 September 2017

No calculators or reference material.

Pledge: I have neither given nor received aid on this exam nor observed anyone else doing so.

Signature:

Name:

Unique name:

1. **(10 pts.)** Build process

   (a) **(3 pts.)** Consider the following makefile definition and rule.

   ```
   CFLAGS = -mcpu=cortex-m3 -mthumb -g3 -O1

   %.o: %.c Makefile
     ${PREFIX}gcc ${CFLAGS} -c -o $@ $<
     ${PREFIX}objdump -D $@ > `basename $@ .o`.lst
     ${PREFIX}nm $@ > `basename $@ .o`.nm
   ```

   If the user has just completed a build, changes the CFLAGS optimization level in the makefile to 3, and types make again, will the source files be recompiled? **Yes** / **No**

   (b) **(4 pts.)** Sometimes a linker adds new labels and code to a program. Here is one such example.

   ```
   __*ABS*0x4000001_veneer:
   ldr     pc, [pc, #-4]   ; 2000001c <__*ABS*0x4000001_veneer+0x4>
   streq   r0, [r0], #-1
   ```

   When a program branches to this label, will the streq instruction be executed? **Yes** / **No**

   (c) **(3 pts.)** Circle the tool with the job of finalizing the offset literals within branch instructions.

   - Compiler
   - Assembler
   - Linker
   - Make

2. **(10 pts.)** Interrupts

(a) **(2 pts.)** Using one sentence, explain preemption.

(b) **(1 pts.)** Using one sentence, explain why it is useful in embedded systems.

(c) **(7 pts.)** You have designed a robot controller with three external interrupts.

- Interrupt A occurs when the robot detects it will soon collide with an object. It is in the highest priority group (priority group 1) and has a subpriority of 2.
- Interrupt B occurs when the robot's battery energy is low. It is also in the highest priority group (priority group 1) and has a subpriority of 3.
- Interrupt C occurs when the robot enters a power saving mode. It is in priority group 2 and has a subpriority of 1.

For each of the following situations, indicate the order in which interrupts are handled by the processor, using "A", "B", and "C".

i. All of the interrupts occur at the same time.

ii. Interrupt C is executing when interrupts A and B trigger at the same time.

iii. Interrupt B is executing when interrupts A and C trigger at the same time

3. **(10 pts.)** MMIO
   Write a C function that takes an integer input, compares the input to a threshold, and turns on a LED if and only if the input is greater than the threshold. The LED is mapped to memory address 0x45001234 and has two states: on and off. The threshold value is an integer with range [0:255] and can be read from the register mapped to memory address 0x4500FFFF. The LED is active-high.

```
void LEDoutput(int val) {




}
```

4. **(10 pts.)** The C implementation of greatest_arr is on this page. The next page contains an assembly version with some lines missing. Fill in the missing lines so that the function works properly and is ABI compliant. The *fill* function takes in an array and places some values into it.

   This is a simple example of allocating an integer on the stack. Note that the stack grows downwards, and the integer is deallocated by adding to the stack pointer.

```
allocate_int:
    sub sp, #4
    mov r0, #2
    str r0, [sp]
    add sp, #4
```

```
void greatest_arr(uint32_t arr1[], uint32_t return_arr[]){
    uint32_t arr2[5];
    int i = 0;

    fill(arr2);
    for(; i < 5; i++){

        if(arr1[i] >= arr2[i]){
            return_arr[i] = arr1[i];
        } else {
            return_arr[i] = arr2[i];
        }
    }
    return;
}
```

```
greatest_arr:
                                        @add missing line

                                        @add missing line

    mov r2, r0     //r2 = arr1
    mov r0, sp     // r0 = arr2 for fill, r1 = return_arr
    push{r0, r1, r2}
    bl fill
    pop{r0, r1, r2}
    mov r3, #0    // r3 = i
    mov r4, #5    // end loop
loop:
                                        @add missing line

    ldr r5, [r2,r0]  // r5= arr1[i]
    ldr r6, [sp, r0] // r6 = arr2[i]
    cmp r5, r6
                                        @add missing line

if:
    str r5, [r1,r0]    //return_arr[i] = arr1[i]
    b endif
else:
    str r6, [r1, r0]    //return_arr[i] = arr2[i]
endif:
    add r3, r3, #1 // i= i+ 1
    cmp r3, r4
    blt loop    // branch if i < 5
    add sp, #20
                                        @add missing line

    bx lr
```

5. **(10 pts.)** Instruction Encoding
Provide the machine code in hexadecimal for the following instructions. See the attached reference. Assume UAL.

   (a) **(3 pts.)** strb r1, [r2, #7]

   (b) **(4 pts.)** ldrh r7, [r2, #100]

   (c) **(3 pts.)** cmp r2, #32

6. **(10 pts.)** Pointer Exercise
Provide the values of the memory locations after this code executes. Assume the initial memory values are zero. Express memory contents in HEX. Accesses are little endian.

```
mov r0, #100
mov r1, 0x12
movt r1, 0xef
movw r2, 0xabcd
str r2, [r0],0
str r1, [r0,2]!
strh r1,[r0,-1]
strb r2,[r0]
```

The following list of addresses are in decimal.

- 100:
- 101:
- 102:
- 103:
- 104:
- 105:
- 106:
- 107:
- 108:

7. **(10 pts.)** Verilog

Create a Verilog module that generates a pulse-width modulated signal. That means the signal will be a square wave for which the amount of time within each period spent high is controlled (modulated).

This module is controlled by writing to two memory-mapped input-output registers. A register at memory address 0x40050000 sets the period of the pulse-width modulated signal (clock ticks per period) and a register at address 0x40050004 controls the pulse width of the pwm signal (clock ticks per pulse).

You can assume that any period or duty cycle values this module receives will be valid. In other words, you do not need to verify that the duty cycle value is lower than the period value.

APB bus read and write timing diagrams are provided on the last page for your reference.

```
module generate_pwm (
  input PCLK,
  input PRESERN,
  input PSEL,
  input PENABLE,
  input [7:0] PADDR,
  output PREADY,
  output PSLVERR,
  input PWRITE,
  input [31:0] PWDATA,
  output [31:0] PRDATA,
  output pwm
)
```

Space for solution to problem Question 7.

Overflow space. We won't look at this space unless you tell us to after the relevant question.

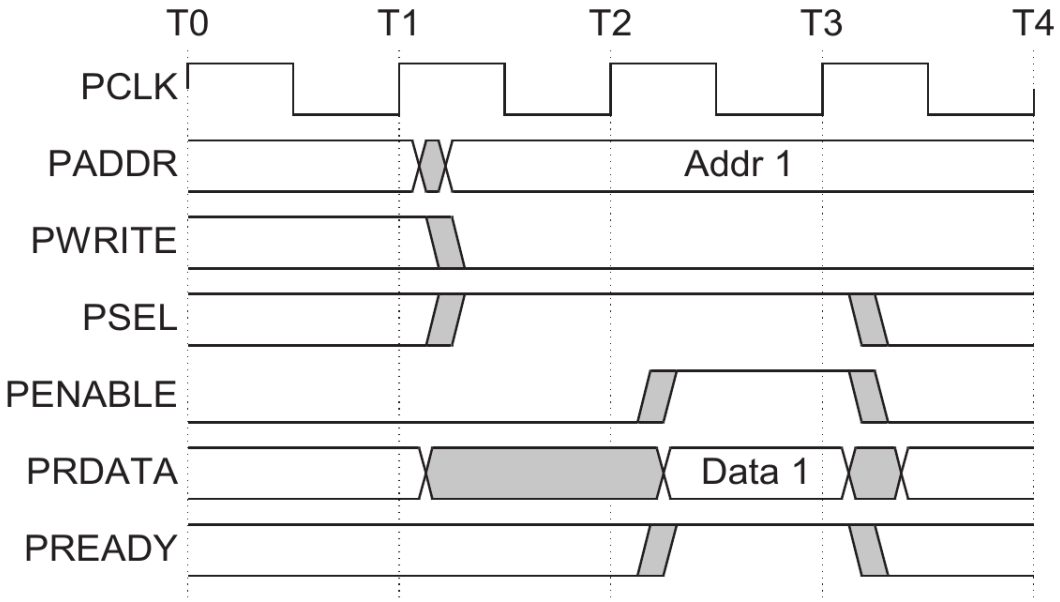


Figure 3-4 Read transfer with no wait states

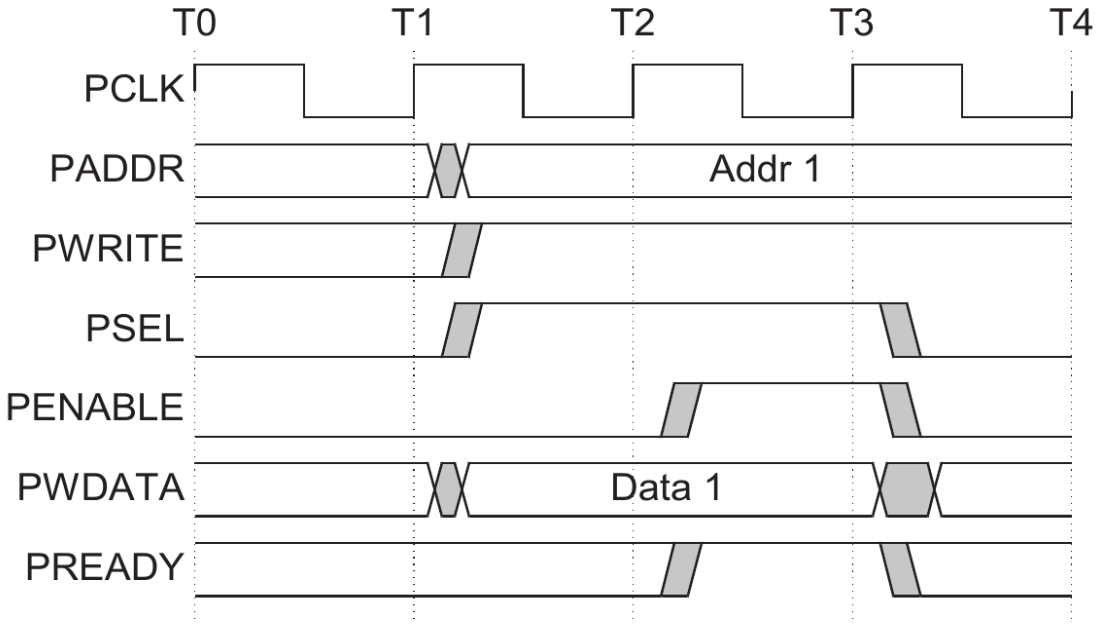


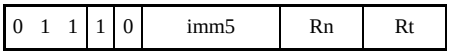
Figure 3-1 Write transfer with no wait states

A6.7.121 STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

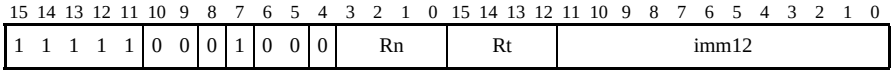
```
STRB<c> <Rt>,[<Rn>,#<imm5>]
```




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv7-M

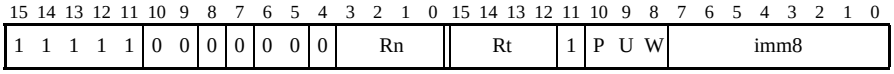
```
STRB<c>.W <Rt>,[<Rn>,#<imm12>]
```




```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t IN {13,15} then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
STRB<c> <Rt>,[<Rn>,#-<imm8>]
STRB<c> <Rt>,[<Rn>],#+/-<imm8>
STRB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```




```
if P == '1' && U == '1' && W == '0' then SEE STRBT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t IN {13,15} || (wback && n == t) then UNPREDICTABLE;
```

Assembler syntax

`STRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE, wback==FALSE`
`STRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE, wback==TRUE`
`STRB<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE, wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. The range of allowed values is 0-31 for encoding T1, 0-4095 for encoding T2, and 0-255 for encoding T3. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>B` is equivalent to `STRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

Exceptions

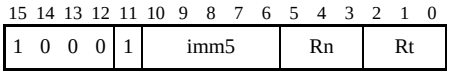
MemManage, BusFault.

A6.7.54 LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

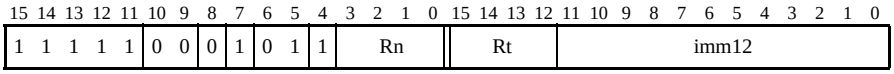
```
LDRH<c> <Rt>,[<Rn>{,#<imm5>}]
```




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv7-M

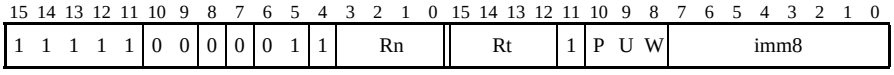
```
LDRH<c>.W <Rt>,[<Rn>{,#<imm12>}]
```




```
if Rt == '1111' then SEE "Unallocated memory hints";
if Rn == '1111' then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
LDRH<c> <Rt>,[<Rn>,#-<imm8>]
LDRH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```




```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Unallocated memory hints";
if P == '1' && U == '1' && W == '0' then SEE LDRHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t IN {13,15} || (wback && n == t) then UNPREDICTABLE;
```

Assembler syntax

`LDRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE, wback==FALSE`
`LDRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE, wback==TRUE`
`LDRH<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE, wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP. If this register is the PC, see *LDRH (literal)* on page A6-112.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 2 in the range 0-62 for encoding T1, any value in the range 0-4095 for encoding T2, and any value in the range 0-255 for encoding T3. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>H` is equivalent to `LDRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

Exceptions

UsageFault, MemManage, BusFault.

Unallocated memory hints

If the Rt field is '1111' in encoding T2, or if the Rt field and P, U, and W bits in encoding T3 are '1111', '1', '0' and '0' respectively, the instruction is an unallocated memory hint.

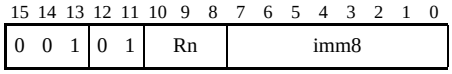
Unallocated memory hints must be implemented as NOPs. Software must not use them, and they therefore have no UAL assembler syntax.

A6.7.27 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb ISA.

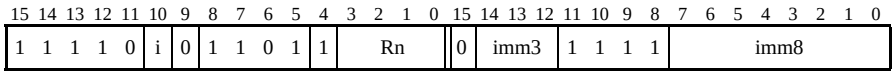
```
CMP<c> <Rn>,#<imm8>
```




```
n = UInt(Rdn);  imm32 = ZeroExtend(imm8, 32);
```

Encoding T2 ARMv7-M

```
CMP<c>.W <Rn>,#<const>
```




```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
CMP<c><q>  <Rn>, #<const>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the operand. This register is allowed to be the SP.

`<const>` Specifies the immediate value to be added to the value obtained from `<Rn>`. The range of allowed values is 0-255 for encoding T1. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values for encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.