538 Lecture Notes Week 2

Announcements

- Labs begin this week.
- Lab 1 is a one-week lab.
- Lab 2 (starting next week) is a two-week lab.

1 Answers to last week's questions

1. If each pixel in an image is 24 bits (8 bits per colour) and the image (or frame) is 400 by 500 pixels, how many frames per second can be transmitted over a system where the address bus is 16 bits, the data bus is 8 bits and the bus frequency is 1 MHz? How many frames can be stored in memory. What would the answers be if the bus frequency were 2 GHz, the data bus and address bus 32 bits each?

ANSWER:

For 1MHz, 8-bit data, 16-bit address:

Bytes per frame = $400 * 500 * 3 = 600{,}000 = 600\,k$

Bytes per second = 1 Megabyte per second

1.66 Frames per second (fps)

With $2^{16} = 65536 = 64\text{K}$ only about one tenth of a frame fits in memory. Conclusion: a 16-bit address bus is insufficient to deal with video.

For 2 GHz, 32-bit data, 32-bit address:

Bytes per frame = $400 * 500 * 3 = 600{,}000 = 600\,k$

Bytes per second = 2G * 4 = 8 Gigabytes per second

$8 \times 10^9 / 600 \times 10^3 = 13{,}333$ Frames per second (fps)

With a memory space of $2^{32} = 4{,}292{,}967{,}296 = 4\text{G}$ we can fit about 7158 frames.

Note:

You may find these numbers astounding. Reality is somewhat different:

i) Video is almost always encoded (MPEG-4 compression reduces the size by a factor of at least 20.) TV requires 30 fps; movies 24 fps. So an hour of video is 108,000 frames; after compression a 2.5 hour movie can fit onto a 4GB dvd.

ii) A 2GHz processor only operates at that speed on a very tiny and expensive *cache* memory. The bus speed to DRAM memory is *much* slower. Fortunately, it is more than fast enough to retrieve compressed video and the processor is fast enough to decompress it in real time.

2. Consider the following program fragment:

```
      org $4000
      lds #$5000  ;load $5000 into S register
      jsr foo
here  bra here

foo   ldaa #2
loop1:
       jsr goo
       deca
       bne loop1
       rts

goo:
     bra done
done:
     rts
```

Translate (manually) the program to machine code and show what the memory dump would look like.

Answer:

```
4000  cf 50 00 16 40 08 20 fe 86 02 16 40 11 43 26 fa 20
4010  00 3d
```

Show the contents of the stack just before the instruction labelled “done” is executed.

Answer:

```
4ff0 uu uu uu uu uu uu uu uu uu uu aa bb 40 0d 40 06
```

How many times is that instruction executed before the program enters an infinite loop?

Answer:

2 times

3. Consider the following memory dump:

```
4000 12 fe 01 ff fc 31 62 43 02 01 11 22 33 44 55 66
4010 ff ff 99 1a bb cc dd ee ff 10 20 30 40 50 60 70
```

If a signed 16-bit integer is stored at location 0x4010, what is its value in decimal?

Answer:

-1

An 8-bit signed integer is stored at location 0x4013. What is its value in decimal? What is its binary representation? What does its negative look like in binary and hex?

Answer:

```
*0x4013 = 0x1a = 0b00011010 (binary) = 26 (decimal)
-26 (decimal) = 0b11100110 = 0xe6
```

Assembly language

- Machines understand machine code; programmers prefer *symbolic assembly language*.
- Each machine code instruction can be expressed symbolically in a manner that is easier for the programmer to remember.
- For example, the machine code 0x42 has the equivalent assembly language instruction `inca` (short for "increment a").
- Taking the "inca" instruction as a template, you might be able to guess what the assembly language instructions "deca" or "incb" do.
- ("deca" is short for "decrement assumulator a"; "incb" is short for "increment accumulator b").
- Note: Yes, there is a second 8-bit accumulator called "b". This week we will only use the 4 registers previously mentioned: PC, S, A and CCR. We will look at all of the registers next week.
- Besides a mnemonic form for machine code, an assembly language program also includes:
    - Symbolic constants
    - Directives to reserve or initialize memory.
    - Symbolic names for variables.
    - Labels to identify the addresses of instructions symbolically.
    - Directives to specify the address of the next instruction or data.

Here is an example of an assembly language program:

```
;Anything after a semi-colon is treated as a comment
CLOCK_MHZ equ 25                  ;Clock frequency in MHZ
CYCLE_NS  equ 1000/25             ;Cycle time in ns
CYCLES_10us  equ 10000/CYCLES_NS  ;Cycles for 10 microseconds
LOOP_CYCLES equ 4                 ;Cycles per loop iteration
N_LOOPS equ CYCLES_10us/LOOP_CYCLES

        org $4000  ;The dollar sign ($) indicates hexadecimal
;The next instruction will be assembled at 0x4000
        jsr delay10us
forever:
        bra forever


;The delay10us provides a software delay of approximately 10 microseconds
delay10us:
       ldaa #N_LOOPS
loop:
       deca
       bne loop
       rts
```

Here is another fragment of assembly language. (This uses some instructions and addressing modes that we will examine in detail next week. In particular, when we say “ldaa #5”, the value of 5 is loaded into A; this is called *immediate* addressing mode. However, when we say “ldaa 5”, then the contents of memory location 5 are loaded into A; this is called *extended* addressing. The “adda” instruction adds its operand to the accumulator. The instruction “adda q” would be first simplified by the assembler to “adda $6001”, meaning add the contents of memory location 6001 to A. Since these contents have been initialized to 5, the instruction “adda #5, using immediate addressing, would have the same effect.)

```
   org $6000
p    ds.b 1 ;reserve 1 byte of memory
q     db   5 ;initialze next memory location with value 5
r     ds.b 1

      ldaa p  ;load a with the value at symbolic location “p”
      adda q  ;Add to a the value at location “q” (will be 5)
      staa r  ;Store the result at location “r”
```

2 Registers

- **Accumulator B** (B): An 8-bit data register like Accumulator A.
- **Accumulator D** (D): A 16-bit data register that is the concatenation of A (most significant byte) and B (least significant byte). For example, if A = 0x12 and B = 0x34, then Accumulator D = 0x1234.
- **Index Register X** (X): A 16-bit register usually used as a pointer. Used extensively in *indexed addressing.*
- **Index Register Y** (Y): A 16-bit register usually used as a pointer. Used extensively in *indexed addressing.*

More about the CCR: The N Z V and C bits

- We only used the Z bit last week.
- (We will learn about the other 4 bits in the CCR later on in the course.)
- To summarize:
  - **N** (Negative) is a copy of the most significant bit an ALU operation. It is used by instructions such as `bmi` (branch if minus) or `bpl` (branch if positive).
  - **Z** (Zero) is the NOR of all the result bits. (Hence it is 0 if any of the bits are 1.) It is used in instructions like `beq` or `bne`.
  - **C** (Carry) is a copy of the carry-out of the most significant bit of the adder (ALU). It is used by instructions like `bcs` (branch if carry set) or `bcc` (branch if carry clear).
  - **V** (oVerflow) is the exclusive-or betwen the carry-in and the carry out of the most significant bit. It is used by instructions like `bvs` (branch if overflow set) or `bvc` (branch if overflow clear).

3 Addressing Modes

Some instructions (such as `inca`) are complete and require no additional information or *operands*. When no operand is required, it is called the *Inherent Addressing* mode.

Most instructions do require one or more operands which are specified using an *addressing mode*.

Relative Addressing

The 8-bit (signed) operand is added to the PC to determine the target address of branch instructions.

Immediate Addressing

- Used for constants.
- The operand is the constant itself.
- Specified with # preceding operand
- Examples:

```
ten equ 10
   ;The following instructions are all equivalent
   ;(machine code: 860a)
    ldaa #ten
    ldaa #10
    ldaa #$a
    ldaa #%001010
```

Extended Addressing

- The operand is the address of where to find the variable.
- Examples:

```
     org    $2000
p    dc.b   $20
q    dc.b    2
r    dc.b    $55
   ;The following instructions are equivalent
   ;(machine code: b62002)
   ;Each instruction copies the contents of 0x2002 to A
   ;so AccA ends up with the value 0x55
    ldaa $2002
    ldaa r
    ldaa p+2
    ldaa q+1
    ldaa 8194

;A symbol like q can be used in both immediate
;and extended addressing modes:
       ldd #q  ;loads 0x2001 into Accumulator D
       ldd q   ;loads 0x0255 into Accumulator D
```

Extended addressing is used to access simple variables. For example, give pseudo C:

```
unsigned byte p, q, r, w;
w = p + q - r + 5;
```

We translate to assembler as follows:

```
 ; reserve memory space for the 4 8-bit variables
p   ds 1
```

```
q   ds 1
r   ds 1
w   ds 1

   ;prform the calculation
   ldaa p
   adda q
   suba r
   adda #5

 ;store the result in space reserved for w
   staa w
```

Indexed Addressing

- Suppose we wish to add 3 8-bit numbers that are in sequential memory locations. One solution:

```
ar  ds 3   ;reserve 3 bytes of memory for the 3 numbers
    ldaa ar  ;load first number
    adda ar+1 ;add second number to Acc A
    adda ar+2 ;add second number to Acc A
```

- We would like to put the `adda` in a loop but the *address* has to change each time through the loop.

An *index register* (X, Y or S) can hold and address pointing to the data we wish to access using *indexed addressing* and the index register can be modified each time through the loop. For example

```
adda 0,x  ;add byte at address pointed to by X to AccA
inx       ;increment X register by 1
```

- In general, indexed addressing is indicated in assembly language with:

```
offset,IndexRegister
```

- The most common offset is 0 (zero). However any constant offset can be used. Accumulator A, B or D can also be used to indicate a variable offset.

Examples:

```
org $2000
dc.b $11
dc.b $22
dc.b $33
dc.b $44

org $3000
ldx #$2000
ldaa 0,x  ;load a with contents of address $2000 which is $11
inx       ;increment x to $2001
adda 0,x  ;add $22 to AccA making it $33
ldd  #$0102   ;load D with 0x0102
ldaa b,x  ;load A with contents of $2001+2=$2003, i.e. $44
```

```
dex       ;Decrement X to $2000
ldaa 1,x  ;load A with contents of $2001 which is $22
```

Pre(post) Increment(decrement) Indexed Addressing

- It is common to use indexed addressing and then increment (or decrement) the index register. For example, we might write:

```
ldaa 0,x  ;load A with byte pointed to by X
inx       ;increment X to the next byte
```

- We can do both operations with one instruction using the auto post-increment addressing mode:

```
ldaa 1,x+
```

- We can post-increment by any number 1—8. For example:

```
ldx #$1000
ldab 5,x+  ;load B with contents of 0x1000 and then
            ;increment X to 0x1005
```

- We can also increment the index register *before* accessing the data it points to.

```
ldy #$1000
ldaa 1,+y ;Increment Y to 0x1001
          ;THEN load A with contents of 0x1001
```

- We can also pre- or post-decrement an index register. For example:

```
ldx #$1000
ldaa 2,-x ;Decrement x to 0x0ffe THEN load A
          ;with contents of 0x0ffe
ldab 1,x- ;Load B with contents of 0x0ffe THEN decrement x
          ;to 0x0ffd
```

Indexed Indirect

- An additional level of indirection is possible with *indexed indirect* addressing mode.
- Typically an array of pointers is set up in memory and an index register initialized to the base address of the array.
- A pointer in the array is then accessed. The memory it points to is the actual operand. For example:

```
org $2000  ;array of pointers
dc.w  $3003
dc.w  $3001
```

```
dc.w  $3002

org $3000
dc.b  $12
dc.b  $34
dc.b  $56
dc.b $78

ldx #$2000  ;load X with base address of pointer array
ldaa [2,x]  ;Retrieve pointer at $2002, i.e.$3001.
            ;Load contents of $3001($34) to Acc A
```

- Accumulator D can also be used (instead of a constant) for the table offset. For example:

```
ldd #2
ldaa [d,x]  ;load A with contents of $3002 which is $56
```

4 Data Movement Instructions

Load and Store instructions

- A memory location can be copied to a register with a *load* instruction.
- For example, `ldaa, ldab, ldd, ldx, ldy, lds.`
- The operand of a load instruction can be specified using immediate, extended or any of the indexed addressing modes.
- Similarly, a register can be copied to memory with a *store* instruction such as `staa, stab, std, sts, stx, sty.` Any addressing mode except immediate can be used to specify the memory location.

Transfer and Exchange instructions

- One register can be copied to another with a *transfer* (`tfr`) instruction.
- If one register is 8-bits and the other 16-bits only the lower 8-bits are copied with *sign extension*.
- For example:

```
tfr a,b  ;copy A to B
tfr x,y  ;copy X to Y
```

- The contents of 2 registers can be swapped with the *exchange* (`exg`) instructions. For example:

```
exg a,b  ;interchange A and B
```

- The *Sign Extend* (`sex`) instruction explicitly extends the sign when an 8-bit register is transferred to a 16-bit register. For example,

```
ldaa #$ff
sex  a,x  ;makes X 0xffff
```

Move instructions

- A byte can be copied from one memory location to another with the *move byte* (`movb`) instruction. For example:

```
movb #5,$2000  ;make contents of 0x2000 05
movb 0,x,0,y   ;move byte pointed to by x to address specified by Y
```

- Similary a *word* (2 bytes) can be copied with the *move word* (`movw`) instruction. For example:

```
movw #$1234,2,x  ;move 0x1234 to 2 bytes starting at
                 ;address specified by 2+X
```

Effective Address Instructions

- When indexed addressing is used, an *effective address* is first calculated and then the contents of that address is manipulated.
- However, the address itself can be obtained (with no memory reference) with the *load effective address* (`leax, leay, leas`) instructions. For example:

```
leax 5,x  ;add 5 to X
leay 3,x; ;set Y = X + 3
```

- Note: only constant offset indexed addressing modes are supported.
- However, Accumulator B can be added to X or Y with the `abx` and `aby` instructions.

5 Arithmetic operations

Adding/Subtracting

- 8-bit or 16-bit signed or unsigned numbers can be added or subtracted in Accumulators A, B or D. For example:

```
adda #4  ;add 4 to Acc A
addd 0,x ;add 16 bit number starting at address X to D
subb #-5 ;Subtrract minus 5 from B
```

- The negative (2's complement) of an 8-bit number can be set with a *negate* (`neg`) instruction. For example:

```
nega   ;Makes A = -A
neg 3,x ;Negates the contents of X+3
```

- There is also *Add with Carry* instructions (`adca, adcb`) that add the carry bit from a previous operation. This is useful for multi-byte precision arithmetic. For subtraction, there are the *Subtract with Borrow* instructions (`sbca, sbcb`).

Is the answer correct?

- When adding 2 8-bit (16-bit) numbers, the answer may not fit inn 8-bits (16-bits).
- Consider 0xff + 0xff: the answer will be 0xfe with a *carry* of 1.
- If the numbers are considered signed (i.e. -1), then the answer (0xfe = -2) is correct.
- However if the numbers are considered unsigned (255), then the answer (254) is wrong.
- The answer is wrong for unsigned numbers if the Carry bit (in the CCR) is set.
- For signed numbers, the answer is wrong if 2 positive numbers yield a negative result or if add 2 negative numbers yields a positive result.
- An incorrect signed result can be detected in hardware by taking the exclusive-or between the carry-in and the carry-out of the most significant bit of the adder. The CCR V (oVerflow) bit is set by the output of this exclusive-or. Consequently, when adding 2 signed numbers, the answer is incorrect if the V bit is set.
- For example:

```
ldaa p_unsigned
adda q_unsigned  ;add 2 unsigned variables
bcs wrong_sum    ;Branch if Carry Set
ldab p_signed
addb q_signed   ;add 2 signed variables
bvs  wrong_sum  ;Branch if oVerflow Set
```

6 Codewarrior and notes on Lab 1

- In-class demo.

7 Vocabulary

New terms are in **bold**.

Memory Dump	The standard way to display the contents of a block of memory in hexadecimal.
CPU	The Central Processor Unit, the computer's heart.
Bus	A set of parellel wires of digital signals
Address Bus	It indicates the address involved in a bus operation
Data Bus	The data being transferred during a bus cycle.
Control Bus	Signals to time and define the type of bus cycle.
IDE	"Integrated Development Environment". Includes editors, compilers, assemblers, disassemblers, etc. We use *CodeWarrior* in this course. In your Java course, you used *Netbeans*. *Eclipse* is another IDE you may use in other courses.
Read Bus Cycle	Data transferred to the CPU.
Write Bus Cycle	Data transferred from the CPU to memory or a memory-mapped device.
Idle Bus Cycle	Bus inactive
Assembler	Software to translate from symbolic assembler to machine code
Disassembler	Software to translate from machine code to symbolic assembler.
Assembler directive	A line of assembler code that gives information to the assembler software and does not correspond to a machine instruction.
Program Counter	A register containing the address of the next instruction.
Stack Pointer	A register containing the address of the top of stack.
Condition Code Register	Contains bits indicating various conditions (such as whether the last number added was zero, negative, generated a carry, etc.) Also called the *Status Register* in some machines (such as Intel processors).
Index Register	A register mainly used as a pointer. It's size equals the width of the Address Bus.
Arithmetic Shift	Only applies to a right shift where the sign bit is "shifted in" from the right maintaining the sign of the shifted value.
Indexed Addressing	Accessing the contents of memory whose address is calculated by adding an offset (usually zero) to an index register.
Indirect Indexed Addressing	Using indexed addressing to obtain another pointer in memory and then dereferencing the location it points to.
Overflow (V) bit	Set when the result of an addition/subtraction will not fit in the number

	of bits used.
Effective Address	The address that will be used in indexed addressing. i.e. the index register + offset.
Addressing Mode	The way an operand is specified.
Inherent Addressing	The operand is inherent in the instruction itself.
Immediate Addressing	The operand is part of the instruction (a specific field) and no further memory reference is required.
PC-Relative Addressing	The operand is an offset relative to the current value of the PC.

8 Questions

1 Write code so that the least significant bit of Accumulator A is cleared, the most significant set and the other 6 bits inverted. For example, 0x7f would become 0x80 or 0b01011100 would become 0b10100010.

2 Convert the following pseudo-C into assembler:

```
byte bytes[] = {2, 7, 1, 8, 3, 1, 4, 8};
int i = 0
int sum = 0
byte * bp = bytes; //same as bp = &bytes[0]
while(bp <= bytes+7) {
      sum = sum + (*bp)*2;
      bp = bp + 2;
}
```

3 Assuming that the NZV and C bits are all zero, show how they evolve with each instruction:

```
      org $3000
      ldaa #0f
      ora $3000 ;Note: some disassembly required!
```

4 Translate the following pseudo-C to assembler.

```
unsigned byte p, q;

while(p >= 128) {
  if(q < 128) {
```

```
      p--;
    } else {
      p = p + 2;
    }
}
```

5 Translate the following C code into assembler:

```
byte p, q, w;
w = p + q + w;
if (w == 0)
      w++;
```

Vocabulary

Memory Dump	The standard way to display the contents of a block of memory in hexadecimal.
CPU	The Central Processor Unit, the computer's heart.
Bus	A set of parellel wires of digital signals
Address Bus	It indicates the address involved in a bus operation
Data Bus	The data being transferred during a bus cycle.
Control Bus	Signals to time and define the type of bus cycle.
IDE	"Integrated Development Environment". Includes editors, compilers, assemblers, disassemblers, etc. We use *CodeWarrior* in this course. In your Java course, you used *Netbeans*. *Eclipse* is another IDE you may use in other courses.
Read Bus Cycle	Data transferred to the CPU.
Write Bus Cycle	Data transferred from the CPU to memory or a memory-mapped device.
Idle Bus Cycle	Bus inactive
Assembler	Software to translate from symbolic assembler to machine code
Disassembler	Software to translate from machine code to symbolic assembler.
Assembler directive	A line of assembler code that gives information to the assembler software and does not correspond to a machine instruction.

CodeWarrior

You can download CodeWarrior from:
http://www.freescale.com/webapp/sps/site/overview.jsp?code=CW_SPECIALEDITIONS .

Choose the CodeWarrior for HCS12(X) Microcontrollers (Classic)