ECS 50 Notes

Lecture 1:

OH: Kemper 3051 Time: TBD

CPU consists of ALU (Arithmetic Logic Unit), Control Unit and special memory location called registers

SLOC: Source lines of code

Complex concepts managed through abstraction, or breaking it down and manipulating details

Lecture 2:

LDA-load into the accumulator

ADA-add to the accumulator

STA: score to the accumulator

HLT-halt

Example:

lda a

ada b

sta c

htl

a: .word $000003

b: .word $000005

c: .word $000000

All Fs is a standout instruction: Halt

Magnetic Disks: can be polarized to represent a 1 or a 0

Memory: each bit can be though of as an electronic switch representing a 0 or a 1

A bit is the fundamental data component, either a 1 or a 0. Eight bits is called a byte so we can get 256 different numbers. Or 256 of anything that can be represented as discrete entities.

Computers use a counting number system of base 2. So the number 1732= 11011000100

Converting between base 2 and base 10:

Convert 1877 to base 2. We get 2^10=1024 and all the values below it. Start subtracting the number from base 2 from the original number. So we get 11101010101. Going backward we get 1024+512+256+64+16+4+1=1877.

Another way to convert:

1877/2=938 R1

938/2=469 R0

469/2=234 R1

234/2=117 R0

117/2=58 R1

58/2=29 R0

29/14=14 R1

14/2=7 R0

7/2=3 R1

3/2=1 R1

1/2=0 R1

Writing it down from bottom to top we get 11101010101.

Practice: 299:

299/2=149 R1

149/2=74 R1

74/2=37 R0

37/2=18 R1

18/2=9 R0

9/2=4 R1

4/2=2 R0

2/2=1 R0

1/2=0 R1

Answer: 100101011

Conversion problem will be on exams.

Practice:150:

150/2=75 R0

75/2=37 R1

37/2=18 R1

18/2=9 R0

9/2=4 R1

4/2=2 R0

2/2=1 R0

1/2=0 R1

We get 10010110

Terabyte=1000 gigabytes

Tebibytes= 1024 gigabytes

Hexadecimal: base 16 counting. Using digits 0-9 and A-F. so 2AF is 2x256+10x16+15x1=687

Octal Numbers use a base 8 for counting

Lecture 3:

3­­10­=00000011

-3­10=11111101 for complement, you add 1 to the end

Adding these we get 00000000

To get the negative, flip the bits:11111111 and add 1 to the end to get 100000000

ASCII-American Standard Code for Information Interchange. Designed to assign a number to each character. Started out as 7 bit, later given an eighth bit. Unicode was developed to alleviate the restriction to the English language that ASCII had. Textbook doesn’t cover Unicode. Cusp use this, from 0 to 127.

Floating point representation: (CUSP doesn’t have floating point)

For a number 937.155, in scientific notation it is 9.37155\*10^2.

6.2510=110.01 in binary. When we normalize, we move the binary point over to the space where there is only one nonzero space to the left of the point. So we get 1.1001\*2^2. If CUSP had FP, it would allocate 16 bits to 1.1001 (or the mantissa) and 8 bits for the exponent, along with the signed bit. For the exponent, we represent by the table where -127 exponent=1=01,

-1=127=7F, 1=127=81, 126=254=FE, 127=255=FF

Since we have the exponent 2, we write 130 in binary as 10000010

So we would get [exponent,sign,mantissa]=[10000010,0,100100000000000]

IEEE 754 Standard for Floating Point (32-bit) gives mantissa 24 bit(1 bit sing/23 for number). Exponent gets 8 bit. It is in the order [sign, exponent, mantissa].

Stored-Program Computer: a computer which stores programs in memory. Modern computers are employing this architecture. The CUSP computer follows the Con Neumann approach.

CUSP memory: 8-bit bytes. 24-bit words

CUSP CPU: control unit tells arithmetic logic unit what operation to perform. Accumulator is the 24-bit register where arithmetic happens. Program counter is the 12-bit register that keeps track of which instruction to run next.

Instruction Set-tells what a computer can do. CUSP is 24-bit pattern.

Lecture 4:

The operand instructions make up one major group of instruction in CUSP. The other is operate.

Mnemonics-used to make instructions more readable

Fetch-Decode-execute cycle: executes instruction in a continuous cycle. Move instruction from memory to program counter. Increment the program counter. Decode then execute the instruction in the instruction register. Load-determine-do.

On the exam, they may ask about assembling by hand.

Problem: Compute the sum of the squares of all integers from 1 to n.

High level(python):

Counter=1

N=5

Sum=0

While counter<= n:

Sum= sum + counter \* counter

Counter = counter + 1

Turning into CUSP-oriented solution:

Load counter into accumulator

If counter > n then end somehow

Multiply accumulator by counter

Add sum to accumulator

Store accumulator to sum

Load accumulator with counter

Add 1 to accumulator

Store accumulator to counter

Go to top of loop

Initialize counter to 1

Initialize sum to 0

Initialize n to some value

Model: code we want to execute then we initialize our values.

Turning into (partially) assembly:

(jgt=jump, cma=compare, jmp=hard jump, mul=multiply, #= immediate address mode, tells to look at the actual values)

Lda counter

Loop: Cma n (comparison sets flags to compare to)

Jgt end

Mul counter

Ada sum

Sta sum

Lda counter

Ada# $001

Sta counter

Jmp loop (there!)

End: hlt

Counter: $001

Sum: $000

N: $005

Last three instructions aren’t quite right. We’ll learn in chapter 4. We can change it to:

Counter: .word $001

Sum: .word $000

N: .word $005

Converting to full assembly language, we add an address in the beginning of each line of code.

To convert to machine language, we convert all the opcodes/addresses to hexadecimals.

In assembly, the “;” means comment. Each line of comment requires a ; symbol to be included in the front or where ever you comment after.

Constants-define constants using the .EQU assembler directive

For example, we can say

.EQU max\_int, 5

At the top, and instead of n: .word $005, we get:

N: .word max\_int(check slides)

We can use .EQU @, $100 to start addresses at a specific number.

Discussion Week 1:

Lda loads values to acc while sta copies value from acc and stores in the variable name.

Lecture 5:

Make sure CUSP knows to start at the beginning of the program and not at the beginning of the register

An instruction- increments the LC by 1

.equ @,nnn- puts the value nnn in the LC

.equ symbol, value- does not change the LC

.word- allocates a word in memory and increments the LC by 1

End of Chapter 4. The exam will cover Chapters 1 through 4.

Lecture 6:

Midterm on Wednesday august 21. Beginning of class, will last 60 minutes. Open book, open note. Bring ID. Sample exams will be posted on Canvas, no solution sets provided

Chapter 1: Binary Digits, Chapter 2: CPU, Chapter 3: Instruction Set, Chapter 4: ,

Index Register(XR)- 23 bit register to be used as an index variable which is added to some memory address, giving a new address offset from the original address.

EQ(equal flag) and LT(less than flag) hold the results of a comparison instruction

LDA- load contents at address into ACC

LDA#- load actual value in to ACC

LDA+- load contents at address offset by XR into ACC

The plus sign offsets the $nnn address by XR. Chapter 5 goes into more detail

LDX-load index register direct (from addr $nnn)

The X refers to the index register.

2D array: since memory is only one dimensional, we use a formula to get a 2D array onto 1D

Index = row index\* number of columns in a row + column index

Arrays can be nested within multiple loops.

Lecture 7:

2D Arrays mapped onto 1D memory by the formula Index = row index\* number of columns in a row + column index.

Correct 2d array will be uploaded later.

Exam will include questions that force you to find errors in code

Selection sort: not very efficient but good enough for small scale sorting. Two pointers, one looking for the smallest value and one that shows how far we have sorted the list/array. Swapping values is very important in sorting algorithms. Each pointer is a variable that is in a loop. Outer loop sorts while the inner loop swaps.

Binary search: assume the array is already sorted. Break the array down into subarrays split by the index in the middle

Discussion Week 2:

$ represents values that need to be in hexadecimal

.word depends on the context

The 24 bit word $ABCDEF in eight byte is AB CD EF. Thus we get 3 bytes to choose from, we need 2 bits from the opcode to encode this.

Program counter stores addresses so they cant be negative. Accumulator can have negative values so it has signed arithmetic

Lecture 8:

How computers handle procedure calls.

Main program —> procedure A —> procedure B

Stack is a container for a sequential collection of data. Data can be added or deleted only at one end. Analogy: pez candy dispenser. Only the “top” of a stack is accessible, so it requires very few operations:

Push(item)=add item to top of stack

Pop()=retrieve item from top of stack.

Each of these changes the stack pointer

Stack can be used to keep track of procedure calls and parameters.

Stack pointer (SP) is a 12 bit register which keeps track of the top of stack as items are pushed on or popped off a stack.

In CUSP, we initialize it to the base address of the stack. When we push something on the stack, CUSP subtracts 1 from the stack pointer and puts that something at the address in the stack pointer. When we pop, the values at the top of stack is stored where indicated and 1 is added to the stack pointer. It will now grow in the direction of low memory addresses.

Stack instructions:

PSHA: push the acc onto stack at stack pointer

PSHX: push index registers onto stack at stack pointer

POPA: pop from stack into acc

POPX: pop from stack into index register

LDS $nnn : load stack pointer direct (from addr $nnn)

LDS# $nnn : load stack pointer immediate

STS $nnn : store stack pointer direct

Stack can replace actual code, which can ruin programs. Stack is used to manage subroutine calls. To get our program to call a subroutine, we use:

JSR $nnn , where $nnn is supposedly the beginning of the subroutine (Ex. Procedure A). After JSR is executed, the program counter is incremented to the address following the JSR. The PC is pushed on the stack and $nnn is loaded into the PC.

RTN returns to the address on the top of the stack is loaded into the program counter. The flow of control resumes with the instruction following the JSR

Global variables shouldn’t be used since it causes the code to be coupled to the calling program. The subroutine can’t be easily reused elsewhere because it relies on specific variable names outside the subroutine.

Setup a frame pointer to store parameters without using global variables. The FP also frees up the SP to do other computations. Gives names to the values on the stack. We create names for the values using the .equ directive. We use the directive to associate a name with a relative position in the frame. The frame pointer holds the top/beginning of the frame on the stack.

The frame is a conceptual convenience. CUSP follows the frame pointer, then you add the offset to find the correct value.

Instructions: LDF, STF, ADF, SBF, CMF, TSF: transfer stack pointer to frame pointer, TFS: transfer frame pointer to the stack pointer, PSHF: push FP on stack, POPF: pop from stack into frame pointer.

Steps for Subroutines:

1. Calling program pushes parameters on the stack
2. Calling program executes a JSR to the subroutine
3. Now were in the subroutine
4. The subroutine pushes the current frame pointer on the stack to remember it because its about to revise the frame pointer
5. The subroutine transfers the current stack pointer to the frame pointer
6. The subroutine does its computational thing
7. The subroutine pops the old frame pointer off the stack into the frame register
8. Subroutine executes a RTN and execution begins with the instruction following the JSR in the calling program
9. Calling program pops the parameters off the stack

Lecture 9:

Start off by writing code for a linear search. Then turn it into a subroutine.

Exam Solutions: wrote them in book by hand.

Lecture 10:

When the calling program executes a JSR to the subroutine, it pushes the return address onto the stack. When you done, you pop the old frame pointer off the stack into the frame register, restoring it to what it was before the subroutine call. After the RTN executes, the return address also pops off the stack.

Frame pointer is the one that needs to be constant

Lecture 13:

Characters represented by 7 bit codes, with the 8th, first bit a 0. If we want to use a user prompt, we can use the .char assembler directive.

String: .equ @, $010

.char ‘hi how are you’, strlen

This binds the number of characters to strlen. To retrieve a character, we use ldc <mem add>. Stc will store the characters to the memory address

Once you create your I/O subroutine, you can easily reuse them since they are so hard to make.

IO is done by ports and theoretically there can be 4096 ports. Ports have 12 bit addresses just like memory. In CUSP a port is only 8-bits wide as opposed to a 24-bit wide memory location.

3 ports were dealing with rn:

$000 – keyboard controller status register, tells us if a character has been typed on the keyboard

$001 – keyboard controller data register, this is where we retrieve the character from the buffer to be used in our program.

$317 – screen controller data register, this is where we write a character that we want to print on the screen

There are also 2 instructions to work with:

INB nnn – read 8 bits from the IO port at nnn

OUTB – write 8 bits to the IO port at nnn

Poll – constantly pinging something to see if it is right. Example in Sept3CharKeyboardIO

Lecture 14:

SIE- sets the IE interrupt flag to 1. CUSP will look out for interrupts but we still need to tell the devices to send interrupts.

CIE – set IE flag to 0. Interrupts are no longer enables

INT nnn – start an interrupt service routine (ISR). Push all the flags + PC on the stack nnn is typically a pointer to the address

IRTN- return from ISR. Restore flags and resume program where you left off.

$FFB is for the timer interrupt

$FFA is for the tape drive interrupt

$FF9 is for the printer interrupt

$FF8 is for the keyboard interrupt (the one we will talk about today)

This is also the pecking order. Sample example.

Conway’s Game of Life:

Starting with a 5 by 5 grid. 1s represent live cells while 0’s represent dead cells

In HW5a, you will write the subroutine. The subroutine is the array manipulation were it applies the set of rules to the array and copies it to another array. Start off by copying the array. Then we can try to apply the rules to the array that’s being copied. Original array should not be altered in any way.

In HW5b, you complete the game. Write the calling program. Initialize test grid. Appendix A section 12 for help with how to read a number and use it as a value in the program. This is the number of generations that you want to print. Can also get the display to show it in order, just for fun. The hard part is that we will be using CUSPs timer to pause between the generations for a given amount of time (I think 10 seconds). During this time, the program should be keyboard sensitive so anytime a key is pressed, it breaks out of the timer and moves onto the next generation.

Lecture 15:

Midterm 2 review along with a quick lecture.

Lecture 16:

After the 1970s, high level code goes to a compiler and then a microprocessor interpreter instead of straight to the cpu after the compiler. This gives some flexibility so we can change how the microprocessor interpreter gets code and sends it to the cpu.

CISC – complex instruction set computer

RISC – reduced instruction set computer

Pipelining – way that computers read code. Ex. Fetch, decode, operand fetch, execute, write back.