CS 458 Notes

June 29, 2017

Contents

1 CS 458 . . . **2**

1.1 Day 1 (May 31, 2017) . . . 2

1.1.1 Historical Survey . . . 2

1.1.2 Von neumann Architecture . . . 3

1.1.3 Integrated Circuits . . . 4

1.1.4 Digital Logic Review . . . 4

1.2 Day 2 (June 05, 2017) . . . 6

1.2.1 RAM . . . 6

1.2.2 Applications of ROM and RAM . . . 7

1.2.3 Timing . . . 7

1.2.4 Chapter 4 - Register Transfer Language (RTL) . . . 8

1.3 Day 3 (June 07, 2017) . . . 10

1.3.1 Instruction Set . . . 10

1.3.2 Instruction Format . . . 12

1.3.3 Instruction Encodings . . . 13

1.3.4 Addressing modes . . . 14

1.3.5 Putting it all together . . . 15

1.3.6 Instruction Execution Cycle . . . 17

1.3.7 Chapter 5 - Processor Design (concept into practice) . . . 17

1.3.8 Review last homework . . . 18

1.4 Day 4 (June 12, 2017) . . . 18

1.4.1 Exam 1 . . . 18

1.4.2 Chapter 5 Continued . . . 19

1.5 Day 5 (June 14, 2017) . . . 21

1.5.1 Assembly to Machine Code . . . 21

1.5.2 Chapter 8: Central Processing Unit (CPU) . . . 23

1.6 Day 6 (June 19, 2017) . . . 24

1.6.1 Chapter 8 Continued 24
1.6.2 Chapter 9 - Pipeline & Vector Processing 26
1.7 Day 7 (June 21, 2017) 27
1.7.1 Exam II 27
1.7.2 Parallell Processing (Chapter 9), continued 27
1.7.3 Chapter 10 (Computer Arithmetic) 30
1.8 Day 8 (June 26, 2017) 33
1.9 Day 9 (June 28, 2017) 33
1.9.1 Chapter 12 - Memory 33

1 CS 458

1.1 Day 1 (May 31, 2017)

- Class discusses single processor system, not Fifth generation.
- Chapter 1 handout
- Discuss the chip manufacturing process

1.1.1 Historical Survey

1. Fst Gen: Vac Tube Tech (1945-1953)
   (a) ENIAC - Electronic Numerical Integration Cmptr
      - Made of 1800 vaccum tubes & 1500 relays
      - Performed 5000 addition or subtraction per second
      - used 1000 bits of core memory
      - power usage: 170kw of power (v hot)
      - wired up for specific computations.
        - It had no programming or operating system.
        - The idea of having a "program stored" in memory is attributed to "von neumann" where and the originators of ENIAC designed the first program-stored computer
2. Second Generation: Transistor Technology (1954-65)
   - Made of transistors
     - Source, Drain, Ground
     - G=0 - open, G=1 - closed

- Vaccuum tube replaced by transistor As a result we see computer becomes smaller, faster, uses less power, & is more reliable (does not fail as often).

3. Third Generation: Integrated Circuits Technology (1965-1980)
   - Also called Microchips or ICs
   - Early ICs allowed a dozen transistors on a single chip
   - Impact was computers became faster, more reliable, smaller, used less power.
4. Fourth Generation: VLSI Technology (1980-now)
   - The trend to include more transistors on the same microchip microchip continued over time for making faster, more reliable, smaller computers that also used less power.
   - This is because each gate/circuit has propagation time (delta t). So placing them closer to one another reduces this delay.
5. Fifth Generation: All non-von neumann parallel archictecture.
   - Examples
     - Multicore Archictecture
     - grid comuting
     - and so on
   - System made up of many processors, memory modules, and I/O devices
   - Not just a simple bus, but an interconnection network
6. Beyond Fifth Generation: Low power processor design
   - power = Ps + Pd
     - V = voltage, F = frequency, C = capacitance, I = current
     - static power = Ps = V*I
     - Dynamic power = Pd = $C*V^2*F$

1.1.2 Von neumann Architecture

- Data line vs control line
- Bus

- Parts
  - CPU
    - Control Unit
      - Has most/all the control signals (control lines), so it directs all the components to do things like load/store/input/output device control
    - ALU (+-*/)
  - Input
  - Output
  - Memory (load/store into CPU)

1.1.3 Integrated Circuits

- (VLSI microchips) Are cut from silicon wafers, circuits arranged in a grid.
- Classes of ICs (sorted by density)
  - SSI (Small Scale Integration): 10-100 transistors on a microchip, 1x1 inch
  - MSI (Medium Scale Integration): 100-1000 transistors on a microchip
  - LSI (Large Scale Integration): 1000 to 10000 transistors on a microchip
  - VLSI (Very Large Scale Integration): More than 10000 transistors on a microchip
  - ULSI (Ultra Large Scale Integration): More and more transistors
- Moore's Law
  - Intel founder - Gordon Moore
  - Stated number of transistors on a microchip doubles every 18 months.

1.1.4 Digital Logic Review

- Logical gates
  - AND F = $A * B$

  - OR F = $A + B$
  - NOT F = ˜A
  - NAND F = ‘$(A * B)$
  - NOR F = ‘$(A + B)$
  - XOR F = $A \oplus B$
  - XNOR F = ‘$(A \oplus B)$
- Circuits
  - Decoders (take binary code set one output bit line according to code)
    - 2x4 means 2 input, 4 output
    - n inputs
    - 2^{n} outputs
  - Encoders (opposidet of decoder)
    - 2^{n} inputs
    - n outputs
    - At any time only one of the inputs is one
  - Multiplexers
    - 2^{n} inputs
    - n select lines
    - one output
  - Demultiplexor
    - Single input
    - 2^{n} outputs
    - n select lines
- Flip-Flops
  - A flip-flop stores one bit of information
  - Types
    - D Flip-Flop
      - D input
      - Q and Q’ outputs
      - CP - clock pulse
      - Clear and inputs: When P=1 then Q=1, When C=1 Q=0

        - T Flip-Flop
            - T Input
            - Q and Q' outputs
            - CP - clock pulse
        - SR Flip-Flop
            - S, R inputs
            - Q and Q' outpus
            - CP - clock pulse
        - JK Flip-Flop
            - J, K inputs
            - Q and Q' outputs
            - CP - clock pulse

- Registers

    - A n-bit register stores a n-bitt data.
    - Types
        1. Register with parallel input & parallel output Loads and stores all at once-
        2. Register with serial input & serial output (shift register) Moves lower bit to higher bit on every clock pulse

- Memory

    - Simply: made up of a decoder that selects the address line from the address registers (AR), in conjunction with data registers (DR)
    - Types
        1. ROM - Read Only Memory
        2. RAM - Random Access Memory

1.2 Day 2 (June 05, 2017)

1.2.1 RAM

- Memory Cell A memory cell stores one bit of information.

- Read/write memory

1. RAM Types

(a) Dynamic RAM

- Denser (more bits per chip)
- Cheaper
- Slower
- Requires refresh logic for periodic refresh. Has a capacitance that needs to be charged periodically. Adds to the slowness.
- Used to make main memory

(b) Static RAM

- Less dense
- Expensive
- Faster
- Used for cache memory

1.2.2 Applications of ROM and RAM

- N.B. DRAM uses one transistor, SRAM uses four transistors.

1. ROM

   - System software
   - Things that won't change over time

2. RAM

   - User program and data is stored here.
   - Things that will change over time

1.2.3 Timing

- Memory Access Time (t_a) - The time between initiation of a memory read or write request to the availability of the data

- Memory Cycle Time (t_c)

- Memory Recovery Time (t_c)

- Memory Bandwidth $w=1/t_c$ (number of memory reads or writes per second)

1.2.4 Chapter 4 - Register Transfer Language (RTL)

- RTL is part of HDL (Hardware Description Language)
- Used to describe a hardware block diagram at the register level (level above logic gate level) (level below system level)
- Hardware Description Languages
  - Verilog HDL
  - VHDL
- To describe a system at register level
  1. Identify its registers and their functions Examples:
     - PC (program counter)
     - SP (stack pointer)
     - et al.
  2. Identify micro-operations performed on those registers Example: PC <- PC+1 (increment)
  3. Identify control functions that initiate those micro-operations Example: control signal (c): PC <- PC+1 When C=1, then increment PC
- Register Transfer modes
  1. Parallel mode P: A <- B (when control signal P=1, then transfer content P=1, then transfer content of B into A.) (a.k.a. mov)
  2. Serial mode Refer to notes
- Logical shift operations
  1. Logical Shift Left Operation L: Shl A When control signal L=1, then do logical shift left of register A
  2. Logical Shift Right Operation R: Shr A When contral signal R=1, then do logical shift right of register A
  3. Rotate left (Circular shift left) operation q: Cil A When control signal q=1, then rotate left the contents of register A
  4. Rotate right (Circular shift right) operation q: Cir A When control signal q=1, then rotate right contents of register A

- Tri-State Device
- Bus Transfer
  - Set of lines between registers
  - Often uses a common set of lines to connect several registers
  - Refer drawing
  - Bus <- Register transfer
    1. Through Tri-State devices
    2. Through multiplexors
  - Register <- Bus transfer
    1. Through Decoders
    2. Through Demultiplexor
- Arithmetic Operations
  - Addition P: EA <- A+B (when control signal P=1, then EA=A+B E is a carry bit register.
- Logical Operations Important: at any time, only one of the control signals can be one, but all can be zero.

  P_1: A<-0 P_2: A<-not(A) P_3: A<-xor(A,B) P_4: A<-A*B
- Memory Operations
  1. Read operation
     - R: MBR<-M or R: MBR<-M[MAR] When read signal R=1, then transfer contens of memory word into MBR.
     - MAR - Memory address register
     - MBR - memory buffer register
  2. Write operation
     - W: M<-MBR or W: M[MAR]<-MBR When W=1, then write into memory word.
- Generation of Control Functions

  Example: Increment Program Counter (PC) when (R=0 and T1=1) or (F=1 and T2=0)

  not(R)*T1+F*not(T2): pc <- pc+1

- Vonn Neumann Architecture

  We talked about the purpose of program counter and how the system loads the word from memory at pc into an instruction register. And then the control unit sends out control signals.

1.3 Day 3 (June 07, 2017)

1.3.1 Instruction Set

- Properties

  1. Complete Instruction set should be complete such that can write an assembly to evaluate any function
  2. Efficient Instruction set should be efficient such that frequently needed instructions can be implemeted by one or few instructions
  3. Compatible Instruction Set should be complete such that old programs can still on on new versions of the processor

- Types In order for ISA to be complete it must implement most of the following instructions. Refer to handout on D2L.

  1. Data transfer instructions Move data from and to memory, between registers
     - move
     - load
     - store
  2. Arithemtic Instructions Perform basic arithmetic operations
     - Add
     - Subtract
     - Multiply
     - Divide
  3. Logical Instructions These instructions perform boolean operations
     - AND
     - OR
     - NOT
     - NAND
     - NOR

        - XOR

    4. Program control Instructions These instructions change the control of flow in the program
        - Branch
            * Unconditional (goto)
            * Conditional (if-else,loop)
        - Subroutine calls
        - Flags
            * Carry
            * Sign Bit
            * Overfloppppw
            * Z-Flag
        - Do loops
        - if/then/else
    5. I/O Instructions
        - Input
        - Output
    6. No-operation
        - wait/delay loops
        - Debugging
    7. Special
        - Conversion
            * translate - translate value based on a table correspondence
            * Convert - convert contents of a word from one to another (packed decimal to binary)

- RISC vs CISC
    - RISC - Reduced Instruction Set Computer
        * Only includes frequently needed instructions
        * Spends less time on decoding
    - CISC - Complete Instruction Set Computer
        * Includes frequently and less-frequently needed instructions
        * Spends more time on decoding, which means more time to run every instruction, which means lower throughput

1.3.2 Instruction Format

- Fields

  1. Op-code Specifies the type of operation
  2. Mode It specifies the addressing mode (immediate, direct, indirect, register, PC-register, relative)
  3. Operand field(s) Gives data or address of the data in memory or a register

- Types

  1. Three-address instruction format Three fields Ex: Op C,B,A; C=B Op A Ex: ADD C,B,A; C=B + A
     (a) Opcode
     (b) Mode
     (c) Destination (C)
     (d) Source2 (B)
     (e) Source1 (A)
  2. Two-address format Two fields Ex: Op B,A; B=B Op A Ex: ADD B,A; B=B + A
     (a) Opcode
     (b) Mode
     (c) Destination/Source2 (B)
     (d) Source1 (A)
  3. One-address instruction format One field Ex: Op A; AC=AC Op A Ex: Add A; AC=AC + A N.B. AC is accumulator
     (a) Opcode
     (b) Mode
     (c) Operand (A)
  4. Zero-address instruction format No fields Get the top two elements of the stack and manipulate them, store back onto stack Ex: Op; SP=SP-1 Op SP Ex: Add; SP=SP-1 + SP N.B. SP is stack pointer
     (a) Opcode

- Stack

    - Stack pointer (SP)
    - SP+/-1 refers to one before the first depending on architecture

- Example X = A+B
    - Three-address ADD X,A,B (ADD|Mode|X|A|B)
    - Two-address MOV X,A (MOVE|MODE|X|A) ADD X,B
    - One-address LOAD A (LOAD|MODE|A) (into AC) ADD B (into AC) STORE X (from AC)
    - Cover zero-address at chapter 8

1.3.3 Instruction Encodings

N.B. N = number of instructions

1. Fixed-length instructions Faster, used in more modern istruction sets
    - Vertical Format n = ceiling($\log_2$(N))

Instruction	Opcode
Add	00
Sub	01
Mult	10
Div	11

      Run through a 2x4 decoder with the outputs the opcode, inputs the bits.

    - Horizontal Format Needs N bits, one bit per instruction

Instruction	Opcode
Add	1000
Sub	0100
Mult	0010
Div	0001

2. Variable-length instructions This instruction decoding takes more time, because it has to decode sequentially.

Instructions	Opcode
Add	1
Sub	01
Mult	001
Div	0001

1.3.4 Addressing modes

1. Implied mode The register/output is implied by the opcode

AC

   Example: INC; AC=AC+1

2. Immediate mode

Opcode	Mode	Data

   Example: Load #5; AC=5

3. Direct mode

Opcode	Mode	Address

   N.B. M[x] means memory contents at address x Example: LOAD ADR; AC=M[ADR]

4. Indirect Mode

Opcode	Mode	Indirect Address

   Ex: LOAD @ADR; AC=M[M[ADR]] So you get an address from memory, then go to that address. Ex: LOAD @5; AC=M[M[5]]

5. Register Mode

Opcode	Mode	Register

   Example: LOAD R1; AC=R1

6. Register Indirect Mode

Opcode	Mode	Register

   Example: Load (R2) ; AC=M[R2]

7. Index Register Mode N.B. Your arch needs an index register (RX). Displacement is added to index register to get memory address.

Opcode	Mode	Displacement

Example: LOAD +DISP(RX); AC=M[RX + DISP] Example: LOAD +5(RX); Ac=M[RX + 5]

8. Index Register Indirect Mode

Opcode	Mode	Displacement

Example: LOAD (DISP(RX)); AC=M[M[RX + DISP]] Example: LOAD (5(RX)); AC=M[M[RX + DISP]]

9. PC Relative Mode Add PC and displacement.

Opcode	Mode	Displacement

LOAD DISP(PC); AC=M[DISP+PC]

10. PC Relative Indirect Mode Add PC and displacement, get memory twice.

Opcode	Mode	Displacement

LOAD (DISP(PC)); AC=M[M[DISP+PC]]

1.3.5 Putting it all together

Example: Design a processor with the following features

- Insructions: ADD, SUB, MULT, DIV
- Addressing modes: register and direct
- Registers: R0, R1, . . . , R7
- Memory size: 1 KBytes

1. Abstract Design Example Identify instruction format to be used (two, one, zero operand instructions). Sppecify the fields, number bits per field, opcodes and codes for modes.

   (a) One-address instruction (13-bit)

Opcode	Mode	Operand
2 Bits	1 Bit	10 Bits

Fixed length vertical format

Instruction	Opcode
ADD	00
SUB	01
MULT	10
DIV	11

Addressing modes	Binary code
Register	0
Direct	1

Registers	Binary Code
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
R7	111

Memory size: 1K words, 1024 words, 2^{10}, 10-bit memory address

Ex: find binary code for following instruction

- ADD R5

Opcode	Mode	Operand
00	0	0000000101

- ADD 5; AC=M[5] + AC

Opcode	Mode	Operand
00	1	0000000101

(b) Two-address (24-bit)

Opcode	Mode	Operand 1	Operand 2
2 bits	2 bits	10 bits	10 bits

- Add R4, R5; R4 = R4 + R5

Opcode	Mode	Operand 1	Operand 2
00	00	0000000100	0000000101

- Add 4, 5; M[4] = M[4] + M[5]

Opcode	Mode	Operand 1	Operand 2
00	11	0000000100	0000000101

(c) Three-address (35-bit)

Opcode	Mode	Operand 1 (Dest)	Operand 2 (Src 1)	Operand 2 (Src 2)
2 bits	3 bits	10 bits	10 bits	10 bits

- Add R3, 5, R1; R3 = m[5] + R1

Opcode	Mode	Operand 1 (Dest)	Operand 2 (Src 1)	Operand 2 (Src 2)
00	010	0000000011	0000000101	0000000001

1.3.6 Instruction Execution Cycle

1. Fetch the instruction
2. Decode instruction
3. Calculate effective address(es)
4. Fetch the operand(s) (data)
5. Execute instruction wih data
6. Decide next instruction address (then go to step #1)

1.3.7 Chapter 5 - Processor Design (concept into practice)

Example design

Registers:

- PC (program counter), 12 bits
- A (address register), 12 bits
- IR (instruction register), 16 bits
- TR (temporary data register), 16 bits
- DR (data register), 16 bits
- AC (general purpose & accumulator), 16 bits
- INPR (input register), 8 bits
- OUTR (output register), 8 bits

One address Instruction format

1 bit	3 bits	12 bits
15	14,13,12	11 to 0
I	opcode	address

Addressing modes:

Direct	0
Indirect	1

Size of memory: 4K words, 4 * 1024 words, 2^2*2^{10}, 2^{12} = 12-bit memory address

Instruction cycle:

1. Fetch cycle
2. Indirect cycle
3. Execute cycle
4. Interrupt cycle (then go to #1)

Questions:

1. What page is this handout on or is it available on D2L (chapter 8)
2. Is this instruction set complete? no, need OR
3. Are we missing some registers? we'll cover them as we get to them
4. Notation of number next to line with line through it? means N number lines corresponding to source and dest hookups

1.3.8 Review last homework

Got handout. Refer to it.

1.4 Day 4 (June 12, 2017)

1.4.1 Exam 1

- Monday, June 19, 2017 (one week for now)
- Ch 4,5,6
- Open book & notes
- No computer

1.4.2 Chapter 5 Continued

1. Example processor continued

   - Brief recap of a 4-bit counter

   (a) Instruction encoding

15	14 13 12	11 to 0
I	Opcode	Address

   i. Memory reference instructions

   - Fixed-length vertical format

Instruction	
And	000
add	001
lda	010
sta	011
bun	100
bsa	101
isz	110
reserved for reg ref and io instructions	111

   ii. Register reference instructions

   - Fixed length-horizontal format

15				
0	1 1 1	b_{11}	... b_1	b_0

Instruction			B_{11}	...	B_0	hex
CLA	0	111	1	0000000000	0	7800
CLE	0	111	0	1000000000	0	7400
...						
HLT	0	111	0	0000000000	1	7001

   iii. I/O Instructions

   - Fixed length-horizontal format

15					
1	111	B_{11}	B_{10}	...	B_0

Instruction			B_{11}	...	B_0	Hex
Inp	1	111	1	0000000000	0	f800
Out	1	111	0	1000000000	0	f400
...						
Iof	1	111	0	0000100000	0	f040

(b) Instruction cycle

   i. fetch

   ii. intdirect

   iii. execute

   iv. interrupt

   i. Examples / continued

   - isz & bun (implement do loop)

     ```
     inp ; ac<- input ctr
     stx x ; use memory x as counter
     loop:
     ...
     isz x
     bun loop
     ```

   - bsa (for calling subroutines/procedures)
     - procedure return address stored just before the procedure body
     - last instruction in procedure body should be bun indirect to the stored address (just before procedure body)
   - interrupt
     - flags
       * Input flag - FGI
       * Output flag - FGO
       * Interrupt flag - R (set by system when FGI or FGO are set, after the fetch cycle)
     - I/o mode of data transfer

       A. programmed i/o Input device set FGI=1 to indicate to processor new data then processor will input data & set FGI=0
       Output device set FGO=1 to indicate to processor new data then processor will output data & set FGO=0

       B. interrupt i/o

       C. Direct Memory Access (DMA) (Chapter 11)
     - levels - not covering our example processor
       * Low priority (interruptable)
       * High priority (typically not interruptable)

- Instructions
  A. ION - turn on interrupt (IEN)
  B. IOF - turn off interrupt (IEN)
- Example (see handout "interrupt handling process")

```
ION ; turn on interrupts
LDA 5
Add 6
STA 7
HLT
```

(c) Deriving Hardware Block Diagram from Hardware Definition Language

- See paper notes

1.5 Day 5 (June 14, 2017)

1.5.1 Assembly to Machine Code

- Two passes
  1. Convert logical addresses to memory addresses
  2. Convert to machine code
- Discussion on binary representation
  - Padding numbers to fit width
  - radix-compliment (here, 2's complement), methods:
    * Take 1's complement then add 1
    * Or, Copy the least significant bit to the first 1 (including that 1), then invert the rest

1. Instruction conversions
   - LDA 004
     - No "I" after the address, so direct mode (0)
     - Opcode (from decoder) is 010
     - Address in Hex, thus 0000 0000 0000 0100
     - Thus machine code is: 0010 0000 0000 0100
   - ADD 005
     - No "I" (0)

  - Opcodde is 001
  - 005_{hex} in 16-bit binary = 0000 0000 0101
  - Thus machine code is : 0001 0000 0000 0101
- STA 006
  - No "I" (0)
  - Opcode 011
  - Address in Hex, thus 0000 0000 0110
  - Thus Machine code: 0011 0000 0000 0110
- HLT
  - "I" is 0
  - Opcode is 111
  - B_{11} thtrough B_1 are 0
  - B_1 is set

2. I/O Operations I/O Modes of Transfer (recap) (refer to 1(b)i)

   (a) Programmed I/O

   - Least efficient
   - Example 1: input data (a): Check input flag FGI and when FGI=1 then skip next instruction

     ```
     ILOOP, SKI       ; (a)
            BUN LOOPI
            INP       ; input data
            STA x     ; Store data in memory
            HLT       ; Done
     ```

   - Example 2: output data (b): keep checking input flag FGO, when FGO=0 then skip next instruction

     ```
     OLOOP, SKO       ; (b)
            BUN LOOPO
            LDA x     ; Get data from mem
            OUT       ; output data
            HLT       ; done
     ```

   - Discussion of PIC18 microarchitecture (PIC18F4X2)
   - Refer to handouts on D2L
     - Chapter 6 Handout #1
     - Chapter 6 Handout #2

   (b) Interrupt I/O

  - Conditions for interrupt
    $T_0' * T_1' * T_2' * \text{IEN} * (\text{FGI} + \text{FGO}) : R \leftarrow 1$
  - As interrupt routine is basically a procedure call, one must save and restore registers

(c) DMA (Ch11)

1.5.2 Chapter 8: Central Processing Unit (CPU)

(Fundamentals)

1. Stack

   - Storage device
   - Made of many registers or memory (implementation detail)
   - Applications
     - Handling subroutine and interrupt calls (supports nested calls)
       * Save return address
       * Save registers
   - Parts of stack
     - Top limit/upper limit/top - other side
     - Lower-limit/bottom - one side
     - Stack pointer (SP) - points to top of the stack
   - Properties
     - LIFO - Last Item in, First Item out
   - Operations on the stack
     - Push: put item on top of stack (and grow the stack)

       $$\text{DR} \leftarrow M[\text{AR}], \text{SP} \leftarrow \text{SP}{+}1, \text{Empty} \leftarrow 0, M[\text{SP}] \leftarrow \text{DR}, \text{IfSP} = \text{upperlimit}, \text{thenfull} \leftarrow 1 \quad (1)$$

     - Pop: get item from top of stack (and shrink the stack)

       $$\text{DR} \leftarrow M[\text{SP}], \text{SP} \leftarrow \text{SP}{-}1, \text{Full} \leftarrow 0, M[\text{AR}] \leftarrow \text{DR}, \text{IfSP} = \text{lowerlimit}, \text{then}Empty \leftarrow 1 \quad (2)$$

     - Add

       $$M[\text{SP}-1] \leftarrow M[\text{SP}-1]+M[\text{SP}], \text{SP} \leftarrow \text{SP}-1, \text{Full} \leftarrow 0 \quad (3)$$

        - Zero-address instruction format
        - Add two topmost items on the stack
    - Call

$$\mathrm{SP} \leftarrow \mathrm{SP}+1, M[\mathrm{SP}] \leftarrow \mathrm{PC}, \mathrm{PC} \leftarrow \mathrm{SUBR} \tag{4}$$

        - N.B. recall one can save registers from caller or in callee
        - Go over some examples
    - Return
- Example stack:

	4	
(sp)	3	3rd item
	2	2nd item
	1	1st item
	0	

- Example stack program:

```
; X=A+B
PUSH A
PUSH B
ADD
POP X
```

2. Other topics in processor design
    - RISC vs CISC Architectures
        - Reduced Instruction Set Computer
        - Complex Instruction Set Computer
    - Handling procedure calls

1.6 Day 6 (June 19, 2017)

1.6.1 Chapter 8 Continued

1. Register Window Concept
    - A fixed set of registers are allocated to each procedure (subroutine). THese registers are distributed into four subgroups as follows:
        (a) High registers. Used for communication with caller procedure into getting parameters

(b) Local registers. Used for local variables

(c) Low registers. Communication with callee procedure for giving input parameters and getting output results. (therefore, low becomed the nested procedure's high registers)

(d) Global registers. Used to hold global variables.

2. Berkeley RISC Uses register windows. SPAC also uses register windows.

   (a) 138 registers total

   (b) Registers partitioned into 8 register windows (can also be informally called "register sets")

   (c) 10 global registers

   (d) 6 High registers

   (e) 10 Local registers

   (f) 6 Low registers. N.B. High and low overlap, with the last low register overlapping with the first high register.

   (a) About the total

       - 8 * 10 = 80 total local
       - 80 + 10 = 90 total of local + global
       - 138 - 90 = 48 left over
       - 48 / 8 = 6 registers left over per register window

3. Some CISC, RISC, and Superscalar processors Ref Table 13.1

   - CISC
     - IBM 370/168 (1973)
     - VAX 11/780 (1978)
     - Intel 80486 (1989)
   - RISC
     - SPARC (1987)
     - MIPS R4000 (1991)
   - Superscalar
     - PowerPC (1993)
     - Ultra SPARC (1996)
     - MIPS R10000 (1996)

1.6.2 Chapter 9 - Pipeline & Vector Processing

1. Parallel Processing
   (a) Pipeline processing
   (b) Memory Interleaving
   (c) Multiple Functional Units
   (d) Vector Processing

2. Pipeline Processing This is about decomposing the task into its subtasks and asigning each subtask to a different segment of a pipeline for execution.

   Input -> Buffer$_0$ -> Segment$_0$ -> Buffer$_1$ -> Segment$_1$ -> ... -> Buffer$_k$ -> Segment$_k$ -> Output

   - Concept: Task-time diagram
   - k-Segment Pipeline - Made of k segments. Input one side, output the other.
   - Pipeline clock cycle

Input->	S$_1$	S$_2$	...	S$_k$	->Output
	t$_1$	t$_2$		t$_k$	

     - $z = \max(t_1, t_2, ..., t_k)$
     - frequency: $f = \frac{1}{z}$
   - Types
     (a) Data-Stream Pipelining
     (b) Instruction Stream Pipelining

   (a) Data Stream pipelining N.B. In general, each pipeline segment may not take equal time.
   Example: $C_i = A_i - B_i$ for $i = 1, 2, 3, 4$ is the same as $A_i + \bar{B}_i + 1$
      i. Pipeline hardware block diagram (ref image)
      ii. Task-time diagram (ref image)
      iii. Pipeline execution time T$_p$ = Pipeline execution time (Time to get first outptu) + (time to get remaining output) $T_p = 3 \cdot t + (4 - 1) \cdot t = 6 \cdot t$

   Compare to non-pipelining. Each operation runs in sequence. T_N = Non-pipeline time $(= 3 \cdot t)(4) = 12 \cdot t$

i. Generalization
   - Pipeline
     - k-segment pipeline
     - Pipeline time $T_p = k \cdot t + (n-1) \cdot k$
     - n = number of tasks
     - First product is the time to get he first output
     - Second product is time to get remaining outputs
   - Non-pipelining
     - $T_N = (k \cdot t) \cdot n$

ii. Performance Metrics

   A. Throughput = $\frac{\text{number of tasks}}{\text{time}}$ Pipeline throughput $w = \frac{n}{T_p} = \frac{n}{k \cdot t + (n-1) \cdot t}$
   Take the limit of w as it approaches infinity. Becomes zero.

   B. Speedup = $\frac{\text{serial time}}{\text{parallel time}} = \frac{\text{non-pipeline time}}{\text{pipeline time}}$ Pipeline speedup $S_p = \frac{T_N}{T_p} = \frac{(k \cdot t) \cdot n}{k \cdot t + (n-1) \cdot t}$
   Take the limit of $\mathrm{S_p}$ approaches infitity, becomes k. So more segments becomes faster.

(b) Instruction Stream Pipelining
   - Recall `execution cycle`.
   - Example: Ideal pipelining

```
LDA x
ADD y
STA z
HLT
```

1.7 Day 7 (June 21, 2017)

1.7.1 Exam II

- Wednesday, June 28
- Chapters 8,9,10
- Open books & open notes

1.7.2 Parallell Processing (Chapter 9), continued

Refer to `parallel processing`

1. Pipelining, continued

   - If you have frequent branches, pipelining doesn't work (as you have to empty the pipeline)

   (a) Pipeline Anomalies
      i. Branch Instructions
      ii. Instruction dependencies
      iii. Resource conflicts
      iv. Segments with unequal execution time

   (b) RISC Architecture For part (C) redefine JUMP instruction such that it does not take effect until the following instruction.

Address	(A) Normal Branch Pipeline Emptied	(B) Delayed Jump Pipeline not emptied	(C) Optimized Delayed Jump Pipeline not emptied
100	`LOAD X, A`	`LOAD X, A`	`LOAD X, A`
101	`ADD 1, A`	`ADD 1, A`	`JUMP 105`
102	`JUMP 105`	`JUMP 106`	`ADD 1, A`
103	`ADD A, B`	`NOP`	`ADD A, B`
104	`SUB C, B`	`ADD A, B`	`SUB C, B`
105	`STORE Z`	`SUB C, B`	`STORE A, Z`
106		`STORE A, Z`	

2. Multiple Functional Units Example: $x = \frac{a+b}{c-d}$

   - break down the operations into ones that can be parallelized

3. Memory Interleaving Two approaches:

   (a) High-order memory interleaving
   (b) Low-order memory interleaving

   (a) High-order memory interleaving
      - Memory is broken up into modules of sequential memory.
      - M$_0$ (highest address: m-1), M$_1$ (highest address: 2m-1), M$_2$ (highest address: mn-1)
      - For high-reliability applications - if one module has defective words, just use different modules.
      - Example: (ref image)

   (b) Low-order memory interleaving
      - For high performance computing
      - Example: (ref image)

4. Vector Processing Uses a SIMD Machine (Single Instruction Multiple Data Stream)
   (a) VA (Vector Add Instruction) $C(I) \leftarrow A(I) + B(I) | I = 1, 2, ..., n$
   (b) Vector Dot Product $C(1) \leftarrow \sum_{I=1}^{n} A(I) \cdot B(I)$
   (c) Vector Compare Compare A(II) with B(I) and set C(I) according for I=1,2,. . . ,n
   (d) V AND (Vector AND) $C(I) = A(I) \wedge B(I) | I = 1, 2, ..., n$
   (e) V MAX (Vector Max) $C(I) = \max[A(I), B(I)] | I = 1, 2, ..., n$
   (f) Vector Merge $C \leftarrow [A(1), B(1), A(2), B(2), ..., A(n), B(n)]$
   (g) Vector search for largest or smallest element

5. Parallel Architecture Categorization (Refer to image)
   (a) SISD (Single Instruction Single Data Stream)
   (b) SIMD (Single Instruction Multiple Data Stream)
      - Multiple ALUs
   (c) MISD (Multiple Instruction Single Data Stream)
      - Ignore
   (d) MIMD (Multiple Instruction Multiple Data Stream)
      - Multiple control units, multiple ALUs

6. Parallel Architecture recap
   - ref image
   - T = running time of a program = compute time + communication time
   - Therefore, communication time can negate the benefit of the parallelizedd tasks

1.7.3 Chapter 10 (Computer Arithmetic)

1. Unsigned Numbers

   (a) Addition of Unsigned Numbers

   - Flowchart $EA \leftarrow A + B, V \leftarrow E$ (V=overflow,E=carry out)
   - Example:

     ```
     A = 5  (0101) -->  0101
     B = 12 (1100) --> +1100
                      1/0001
                      ^ ^^^^

                   E -/ \\\\-- Sum
     V=E, V=1
     ```

   (b) Substraction of Unsigned Numbers

   - HDL
     - $EA \leftarrow A + \bar{B} + 1$
     - (If E=0, A<B, else A>=B)
     - $\bar{E} : A \leftarrow \bar{A} + 1$ (Negate A)
   - Example

     ```
     A=5 (0101) ------->  0101
     B=2 (0010) --2's--> +1110
                        1/0011
     ```

2. Signed Numbers

   (a) Representations

   i. Signed magnitude (number with sign bit)
      - N= +5 = (101) = 0101 (leading bit is sign)
      - N= -5 = (101) = 1101 (leading bit is sign)

   ii. 1's Complement (nonnegative left as is, with 0 sign bit, negative is 1's complement with sign bit set)
      - N= +5 = (101) = 0101 (leading bit is sign)
      - N= -5 = (101) = 1's comp = 1010 (leading bit is sign)

   iii. 2's complement
      - N= +5 = (101) = 0101 (leading bit is sign)
      - N= -5 = (101) = 1011

   (b) Addition and Subtraction of Signed Magnitude Numbers

   - (ref p 347)

- Example:

  ```
  +3+(-5)=-2
  A=+3
  B=-5
  AC= 0 011
   AS A
  BR= 1 101
   BS B
  A=(011) ------->  011
  B=(101) --2's--> +011
                purely
  ```

(c) Addition and Subtraction of signed 2's compelement numbers

- Add
  i. $AC \leftarrow AC + BR$
  ii. $V \leftarrow C_s \oplus C_r$
- Subtract
  i. $AC \leftarrow AC + \bar{B}R + 1$
  ii. $V \leftarrow C_s \oplus C_r$
- Add example

  ```
              C_s --\/-- C_r
  +3+(-5)=?         vv
  AC=+3=(0011) -->  0011
  BR=-5=(1011) --> +1011
                  0/1110
  V= C_s XOR C_r = 0 XOR 0 = 0
  ```

(d) Multiplication of signed magnitude numbers

- (ref p 351)
- Example:

  ```
  (+3)x(-5)=-15
   X_s -\ ///-- X
        v vvv
  BR=+3=0 011
  QR=-5=1 101
  ```

```
AC= 0=0 000
SC= 3=_ 011
```

3. Floating Point Numbers

   - $N = m \times 2^e$
   - m is mantissa (typically respresented in signed magnitude)
   - e is exponent (unsigned number)
   - Conversion of number (N) into floating point format
     (a) Convert N into binary
     (b) Normalize N
     (c) Represent N in IEEE Single-precision (32-bit) or IEEE Double-precision (64-bit)
   - Biased representation. If signed numbers are in the range of -N to `+M`, then add a bias (`+N`) to each number to convert the range from 0 to N+M unsigned numbers. (Basically, instead of using a signed exponent, we map the negative-most value to 0, and increment up, in an unsigned fashion)
   - IEEE Single Precision 32-bit format "Hidden 1", leading 1 in normalized mantissa

1	8	23
Sign Bit	E	F

   - Example: Convert N= +18.375 to floating point format
     (a) Convert N to binary

```
18=16+2
  = 2^4+2^1
  = (10010)

0.375 = 0.25 + 0.125
      = 1/4 + 1/8
      = 2^-2 + 2^-3
      = (0.011)

18.375 = (10010.001)
```

     (b) Normalize N $N = +1.0010011x2^4$
     (c) Represent as IEEE Single Precision Floating Point

```
E = bias + e
  = 127  + 4
  = 128 + 2 + 1
  = 2^7 + 2^1 + 2^0
  = (10000011)

S    E                F
0 10000011 00100110000000000000000
```

1.8 Day 8 (June 26, 2017)

1.9 Day 9 (June 28, 2017)

1.9.1 Chapter 12 - Memory

The Two problems

1. Memory is not fast enough
2. There is not enough memory

1. Definitions
   (a) Memory Access Time - The time takes to do a memory read or write
   (b) Memory cycle time - memory access time plus memory recovery time
   (c) Memory recovery time - time between access times
   (d) Memory bandwidth (w) - $\frac{1}{\text{memory cycle time}}$ = Number of reads or writes per second
   (e) Locality Reference - memory addresses generated by a program tend to be confined within small regions of address space, because of do-loops in programs.

       Example: Consider a program of three components, three do-loops. You may not need the other components in memory while the other one runs, vice versa.
   (f) Memory Hierarchy
       i. Secondary Memory (disk,ssd,etc)
       ii. Main memory
       iii. Cache

Comparison:

	Cache	main Memory	Disk
Speed	Fastest (1 ns)	Medium (10 ns)	Slowest (1 ms)
Size	Smallest (1 K)	Medium (1 M - 1 G)	Largest (1 G - 1 T)
Cost	Most expensive	Medium	Cheapest

(g) Hit Ratio $h = \frac{\text{Number of hits to cache}}{\text{Number of hits to cache+Number of misses to cache}}$

Consider: $t_a = h \cdot t_c + (1-h) \cdot (t_c + t_m)$, where $\mathrm{t_a}$ is average memory access time, $\mathrm{t_c}$ is memory cache access time, h is hit ratio for cache, and $\mathrm{t_m}$ is main memory access time

Same as: $t_a = t_c + (1-h) \cdot t_m$

Example: $t_c = 26\text{ns}, t_m = 312\text{ns}, h = .90$

$t_a = 26 + (1 - 0.9) \cdot 312 = 57.2\text{ns}$

2. Multi-level memory hierarchy

   - $\mathrm{s_i}$ = size at level i
   - $\mathrm{t_i}$ = access time at level i
   - $\mathrm{h_i}$ = hit ratio at level i - N.B. is 1 at the highest tier of memory (such as disk), so it's often omitted in equations/examples
   - $s_1 < s_2 < s_3 ... < s_i < s_{i+1} < ... < s_n$
   - $t_1 < t_2 < t_3 ... < t_i < t_{i+1} < ... < t_n$
   - Effective memory access time at level i = emat$_i = t_i + (1 - h_i) +$ emat$_{i+1}$

```
    CPU         s_1, h_1, t_1
     |
Cache Level 1   s_2, h_2, t_2
     |          .
Cache level 2   .
     |          .
Cache level 3   s_i, h_i, t_i
     |
Main memory     s_i+1, h_i+1, t_i+1
     |
   Disk         s_n, h_n, t_n
```

3. Cache-write policy

   (a) Write through policy: update main memory, copy data when the cache is updated

   (b) Copy back: Update main memory, copy data at the time of cache miss

4. Cache organization

   (a) Associative memory

       - Argument register (4 bit)
       - Key register (a.k.a. mask register) (4 bit)
       - Memory
       - Match register (1 bit per word of memory)

       Put data in argument register. Put bitmask in key register. Match register is updated to reflect which words meet the AND of Argument and Key registers as equal.

       Because the matches are done in parallel, logic is necessary for every memory word to facilitate this. Which increases the footprint and cost of associative memory.

   (b) Direct mapping Ref handout.

       Memory access time $\cong \sqrt{\text{memory size}}$

       - CPU address register (two parts, few bits called tag starting at MSB, index is the rest of the register)
       - Cache is a series of words with extra bits for tag. Address corresponds to Index.
       - Entire address register refers to an address in memory. Index refers to just the cache address. Tag is used to verify the index is for the same tag.

   (a) Set-associative mapping (extension of #2)

       Example:

       Set size K=2. Has two "sets", that is, two sections of cache, with tag and data in each word, and address referring to an index. So in effect, it's just Direct Mapping with multiple mappings (which presumably has circuitry to prevent duplicate data)