CISC 260 Machine Organization and Assembly Language

Fall 2021

Review2

Course Catalog Description:

Introduction to the basics of machine organization. Programming tools and techniques at the machine and assembly levels. Assembly language programming and computer arithmetic techniques.

CISC260 Tentative Schedule (F21)

#	Week	Topic
1	Aug31	Overview, Data representation and arithmetic
2	Sept6	Boolean logic, gates
3	Sept13	Build a simple computer
4	Sept20	ARM, ISA, Assembly Language
5	Sept27	Assembly programming
6	Oct4	Procedure call, stack
7	Oct11	Assembly programming and Review1

Midterm Oct 14

#	Week	Topic
8	Oct18	Floating point
9	Oct25	Assembly programming: Dynamic data structure
10	Nov1	Assembler, Linker, Compiler
11	Nov8	Performance and optimization
12	Nov15	Cache
13	Nov22	Thanksgiving (No classes)
14	Nov29	I/O and security
14	Dec6	Review2

2021 Fall Semester Final Exams matching 'cisc260'

If you have questions, contact the Office of the Registrar by writing to schedoffice@udel.edu. Changes to the final exam schedule occur frequently. Final exam schedules should be confirmed with instructors before making travel arrangements. More information on final exams can be found on the Registrar's page ⤤.

Course Number	Exam Day	Exam Date	Exam Time	Exam Location
CISC260011	Tuesday	Dec 14	10:30 AM-12:30 PM	GOR102 ⤤
CISC260010	Wednesday	Dec 15	8:00 AM-10:00 AM	GOR102 ⤤

The final exam will be open-book and open-note, like the midterm exam. No electronic devices are allowed, except for a calculator.

Reading:

Chapter 1.1 - 1.5
Chapter 2.1 - 2.4, 2.8;

Chapter 3.1; 3.2.1 - 3.2.4;

Chapter 5.1 - 5.3
Chapter 6.1 - 6.9
Chapter 7.2, 7.5
Chapter 8.1 – 8.3
Chap 9.1 – 9.2 (online)

Major topics covered include:

- Digital representation of information: decimal, hexadecimal, binary, ASCII
- Arithmetic in binary, two's complement
- Combinational and sequential logic, ALU
- Control and datapath
- Instructional Set Architecture (ISA)
- Machine language and assembly language
- Stacks and procedure calls

The students should be able to

- explain the basic organization of a classical von Neumann machine and its major functional units
- explain how machine code is formatted/organized and executed via the corresponding functional units
- write simple assembly language program segments
- demonstrate how fundamental high-level programming constructs, such as loops, procedure calls and recursions, are implemented at the machine and assembly language level
- convert numerical data between different formats
- carry out basic logical and arithmetic operations

Big ideas:

1. Computing / information processing: $y = F(x)$
2. Universality: All boolean functions can be implemented by wiring a bunch of NAND gates.
3. ALU is programmable (no hard wiring is necessary for a given F)
4. Sequential logic can hold states (memory)
5. Stored programs (von Neumann architecture)
6. Turing complete: Sequence, Branch, and Loop
7. Code reusability and Abstraction: procedures/subrountines
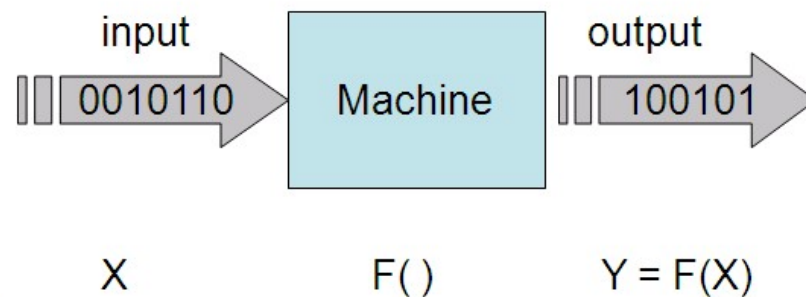8. Stack and recursive calls

What this course is about?

- It is about the *inner* workings of a modern computer

What is computing?
- arithmetic calculating
  e.g., 3 + 2 = 5
- manipulating information
    information? (symbols and interpretation)
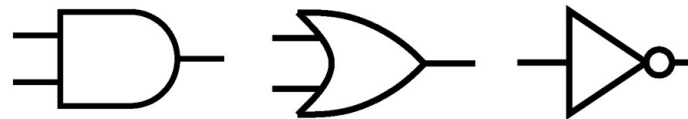   syntax   semantics

input		output
0010110	Machine	100101
X	F( )	Y = F(X)

Any function can be implemented in Boolean logic

Function is a mapping from input variables `I` to
output value `O`.

$$F: I \rightarrow O, \quad \text{where } I \in \{0,1\}^N, O \in \{0, 1\}^M.$$

Inputs ABC	Output XY
000	01
001	00
010	00
011	10
100	10
101	00
110	11
111	00

X=~A&B&C | A&~B&~C | A&B&~C

Y=~A&~B&~C | A&B&~C

CISC260, S07, Li Liao

Build a computer

More layers …

Instruction Set Architecture (ISA):

31	26 25	21 20	16 15	11 10	6 5	0
000000	00010	00011	00001	00000	100000
op	rs	rt	rd	shamt	funct
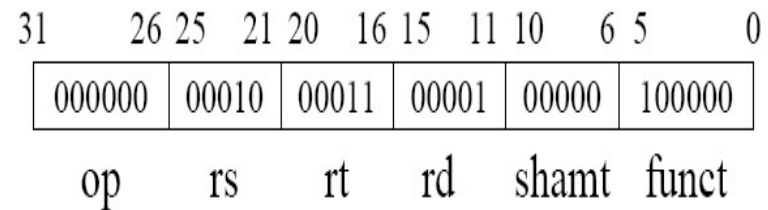
Functional units:

Gates (CPEG 202):

$A+B$ $A \cdot B$ $\overline{A}$
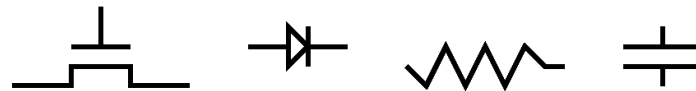
OR   AND   NOT
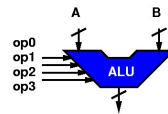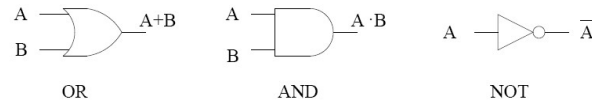
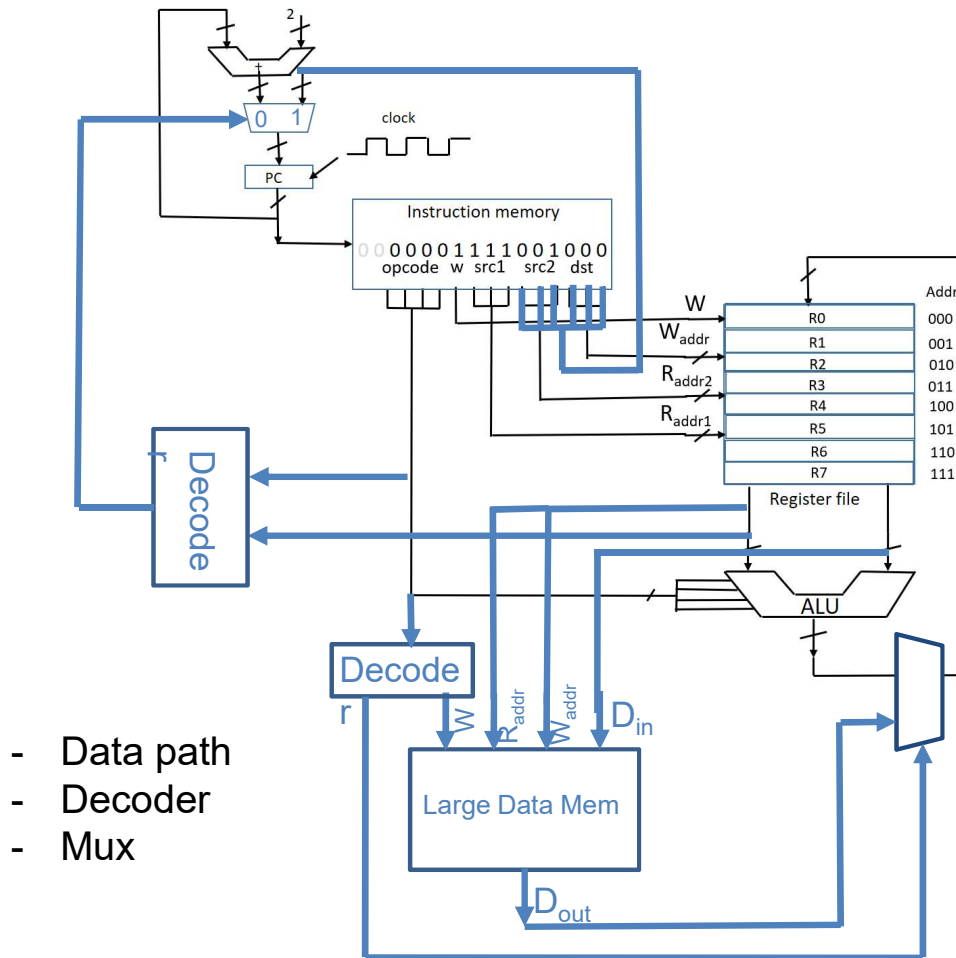Devices (in silicon)

We take a bottom-up approach, starting with gates

A Simple Computer with added large memory

- Data path
- Decoder
- Mux

Data processing instructions:
ADD
SUB
ADD
OR
INV
SLL
SRL
…

Branch instructions:

BRZ R3, LOOP
00 1001 0 011 000010
unused opcode w src BAddr

Data transfer instructions:
ST src1, src2
LD src1, dst

CISC260, Liao, U of Delaware

ARM (32bit, single cycle computer)

- Data path
- Parse/Decoder
- Control/Mux
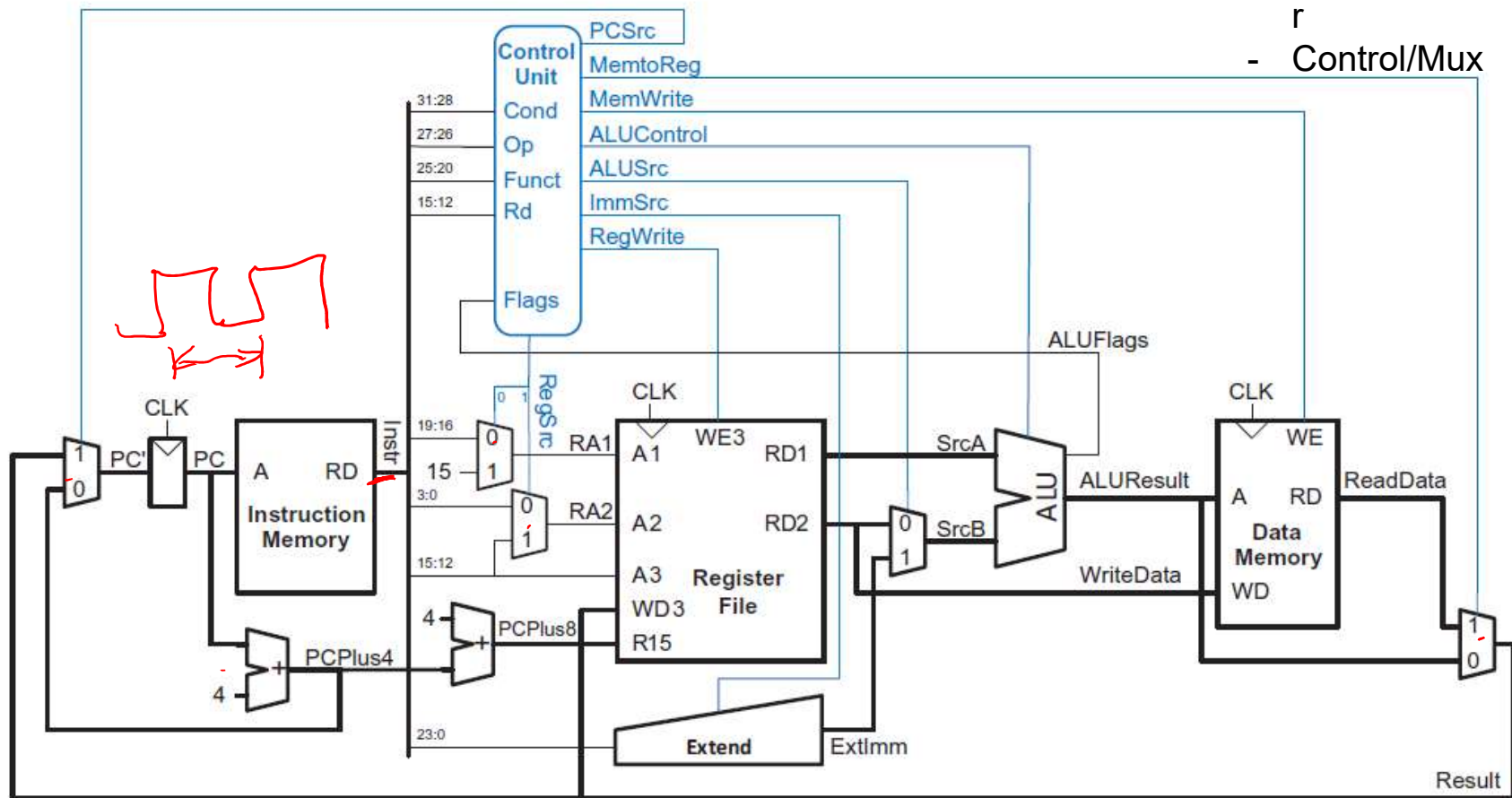
Figure 7.13 Complete single-cycle processor

Liao, CISC260

A Programmer's Perspective

Memory

Address | Segment

0xFFFFFFFC — Operating System & I/O
0xC0000000
0xBEFFFAE8 — Stack ← SP

Dynamic Data

↑ Heap

Global Data

← SB

Text

)00

Exception handlers

0x00000000 ← PC

Figure 6.30 Example ARM memory map

Stored program

Address	Instructions

Assembly code | **Machine code**

```
MOV    R1, #100           0xE3A01064
MOV    R2, #69            0xE3A02045
CMP    R1, R2            0xE1510002
STRHS R3, [R1, #0x24]    0x25813024
```

Address	Instructions
0000800C	2 5 8 1 3 0 2 4
00008008	E 1 5 1 0 0 0 2
00008004	E 3 A 0 2 0 4 5
00008000	E 3 A 0 1 0 6 4 ← PC

Main memory
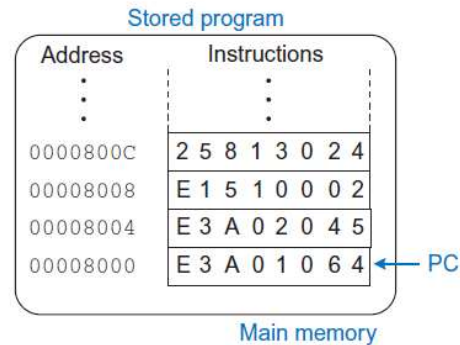
Figure 6.28 Stored program

CPU

Load/Store

registers

ALU

Basic ARM Instructions

Data processing:		Meaning
ADD	r1, r2, r3	r1 ← r2 + r3
SUB	r1, r2, r3	r1 ← r2 - r3
AND	r1, r2, r3	r1 ← r2 & r3
ORR	r1, r2, r3	r1 ← r2 | r3
MVN	r1, r2	r1 ← ~ r2
LSL	r1, r2, #10	r1 ← r2 <<10
LSR	r1, r2, #10	r1 ← r2 >>10

Data Transfer (Memory):

LDR	r1, [r2, #20]	r1 ← Memory[r2 + 20]
STR	r1, [r2, #20]	Memory[r2 + 20] ← r1
SWAP	r1, [r2, #20]	r1 ↔ Memory [r2+20]
MOV	r1, r2	r1 ← r2

Branching:

CMP	r1, r2	Condition flag NZCV is set per the outcome of r1-r2
B	label	PC ← Address of label
BL	label	PC ← Address of label, r14 ← PC+4

The Program Status Registers (CPSR and SPSRs)

31			28																				8			4				0
N	Z	C	V																				I	F	T	Mode				
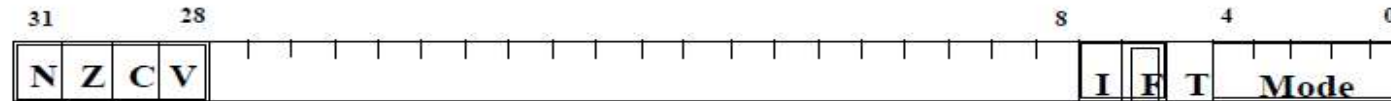
Copies of the ALU status flags (latched if the instruction has the "S" bit set).

* **Condition Code Flags**

  N = **N**egative result from ALU flag.
  Z = **Z**ero result from ALU flag.
  C = ALU operation **C**arried out
  V = ALU operation o**V**erflowed

* **Mode Bits**

  M[4:0] define the processor mode.

* **Interrupt Disable bits.**

  I = 1, disables the IRQ.
  F = 1, disables the FIQ.

* **T Bit** **(Architecture v4T only)**

  T = 0, Processor in ARM state
  T = 1, Processor in Thumb state

ARM Programming

➢Branch (cmp, beq, bne)

♠ "If-then-else"

Example:

```
if(i==j)   f = g +h;
else       f = g – h;
```
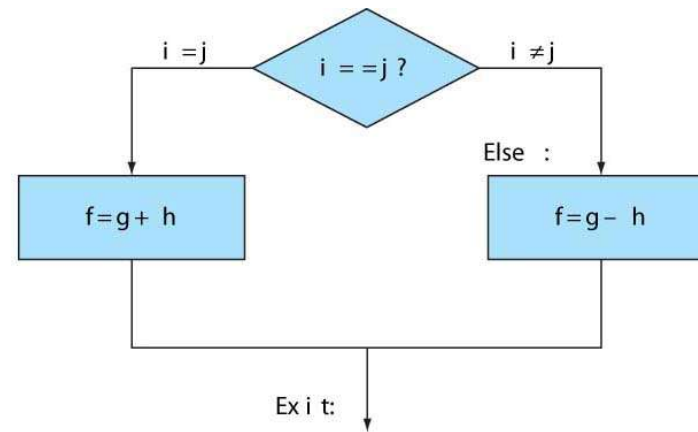
assume f, g, h, i, and j are in r0, r1, r2, r3, and r4 respectively

```
        CMP     r3, r4
        BNE     Else
        ADD     r0, r1, r2
        B       Exit
Else:   sub     r0, r1, r2
Exit:
```

A more compact and efficient version:

```
CMP  r3,r4
ADDEQ r0,r1,r2 ; f = g + h (skipped if i ≠ j)
SUBNE r0,r1,r2 ; f = g - h (skipped if i = j)
```

i =j ← i ==j ? → i ≠j

Else :

f=g+ h

f=g− h

Exit:

Loop

Example:
        while (save[i] == k)
                i += 1;

assume i is in r3, k is in r5, and the base of the array is in r6.

```
Loop:   ADD     r12, r6, r3, LSL #2
        LDR     r0, [r12, #0]
        CMP     r0, r5
        BNE     Exit
        ADD     r3, r3, #1
        B       Loop
Exit:
```

Six steps

1. (caller) place parameters in a location where the procedure (callee) can access them (r0, r1, r2, r3)
2. transfer control to the procedure. (**BL**)

----

caller

3. acquire the storage resources needed for the procedure.
4. perform the desired task
5. place the result value in a place where the caller can access it.
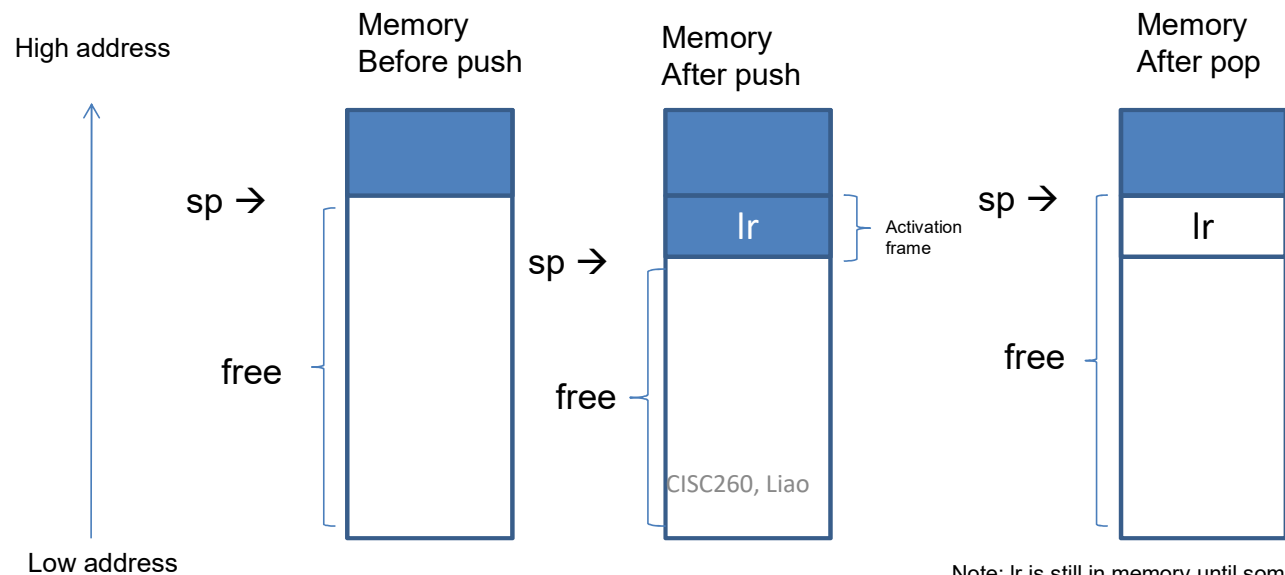6. return control to the point next to where the program is called. (**MOV pc, r14**)

callee

ARM conventions:
- r0 - r3, r12: registers for storing arguments or scratch registers to used by the callee (not preserved).
- r4-r11: registers that need to be preserved, if used by callee.
- lr: register storing the return address (r14)
- sp: stack pointer (r13)

Subroutine: use stack to maintain activation frames for subroutine calls

```
SUB       sp, sp, #4              @ push
STR       r14, [sp]              @ push
…
…
…
LDR       r14, [sp]              @ pop
ADD       sp, sp, #4             @ pop
MOV       pc, r14
```

High address

Low address

Memory
Before push

sp →

free

Memory
After push

sp →

lr   } Activation
                      frame

free

CISC260, Liao

Memory
After pop

sp →

lr

free

Note: lr is still in memory until some new value is
written in the same location. Potential security
loophole.

IEEE Floating-Point Format (Single Precision)

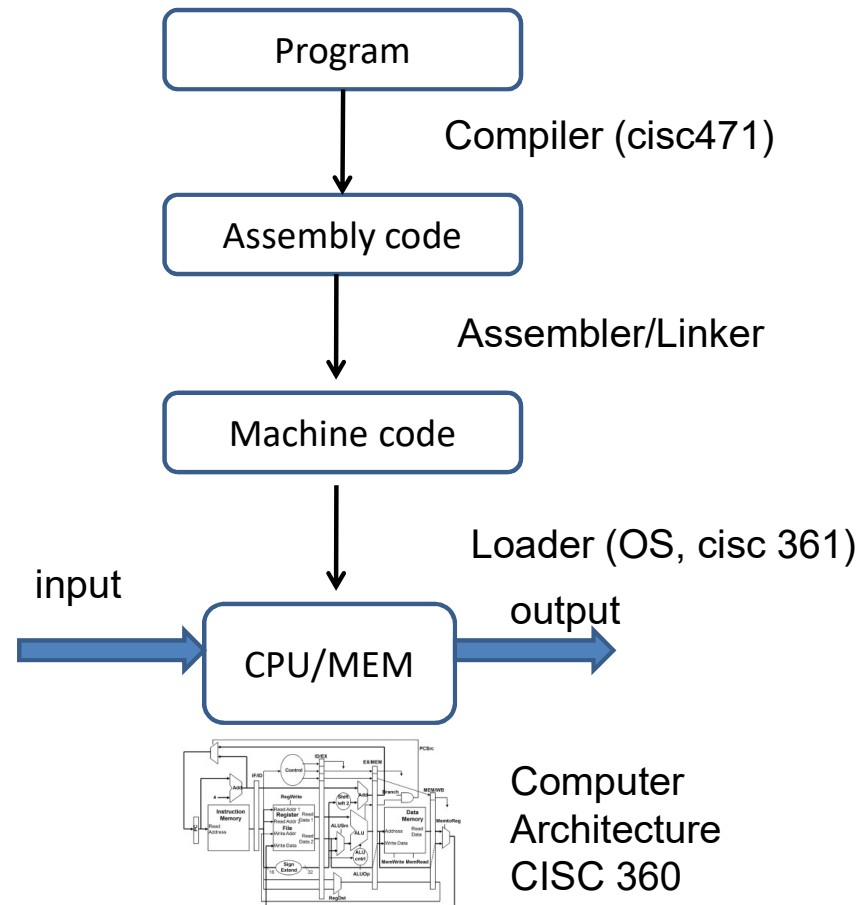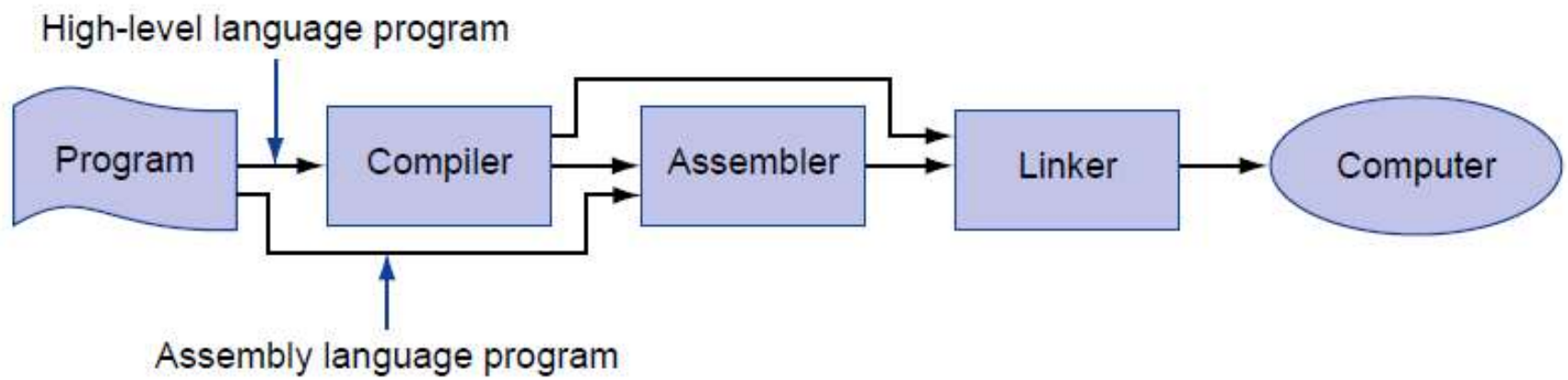| 8 bits	23 bits

S	Exponent	Fraction

[-127, 128]

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

(0-255)
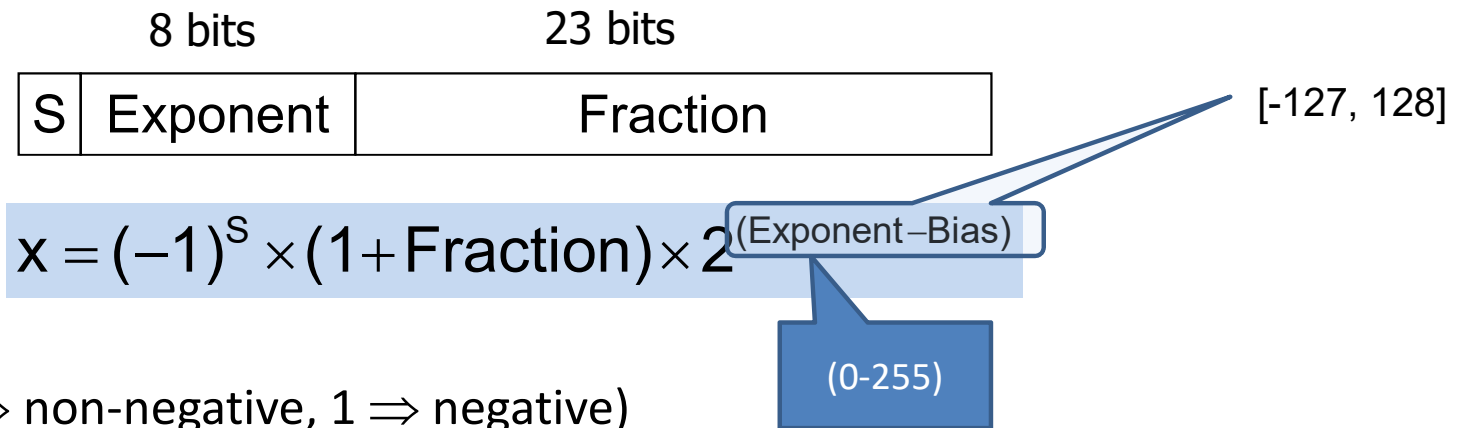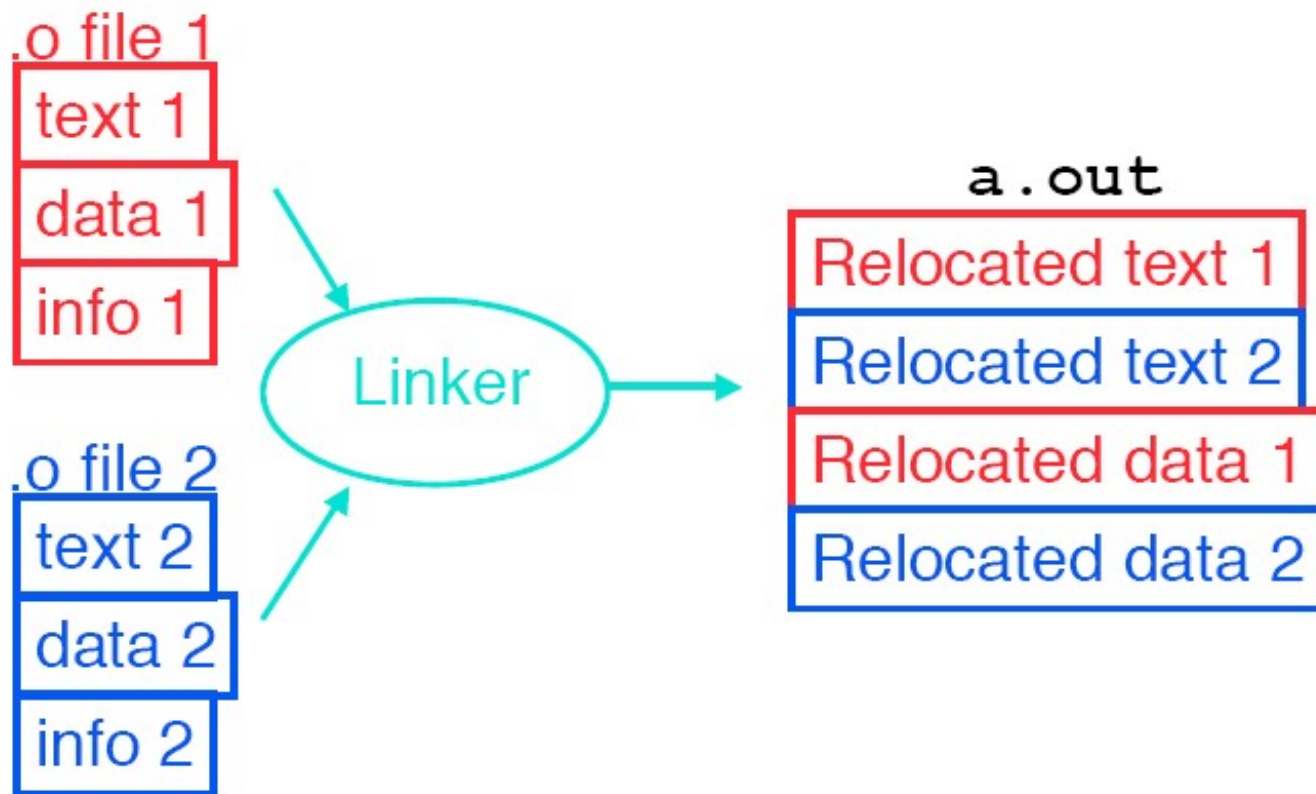
- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalize significand: 1.0 $\leq$ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127

```
              ┌─────────────────────┐
              │      Program        │
              └─────────────────────┘
                        │
                        │     Compiler (cisc471)
                        ▼
              ┌─────────────────────┐
              │    Assembly code    │
              └─────────────────────┘
                        │
                        │     Assembler/Linker
                        ▼
              ┌─────────────────────┐
              │    Machine code     │
              └─────────────────────┘
                        │
                        │     Loader (OS, cisc 361)
    input               ▼            output
  ══════════▶ ┌─────────────────────┐ ══════════▶
              │      CPU/MEM        │
              └─────────────────────┘
```

Computer
Architecture
CISC 360

High-level language program

Program → Compiler → Assembler → Linker → Computer

Assembly language program

.o file 1
text 1
data 1
info 1

.o file 2
text 2
data 2
info 2

Linker

a.out
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

cisc260, Liao

Performance

CPU time = IC x CPI x CC

Amdahl's Law:

$$= \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

where the numerator represents the Execution time after improvement.

Technology	Cost/GB	Access Time
SRAM	~ $10,000	~ 1 ns
DRAM	~ $100	~ 100 ns
Hard Disk	~ $1	~ 10,000,000 ns
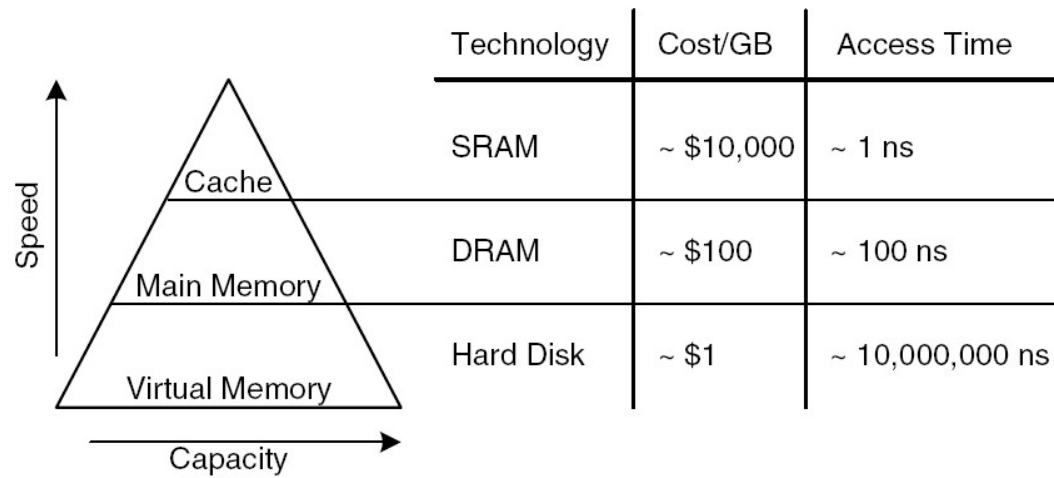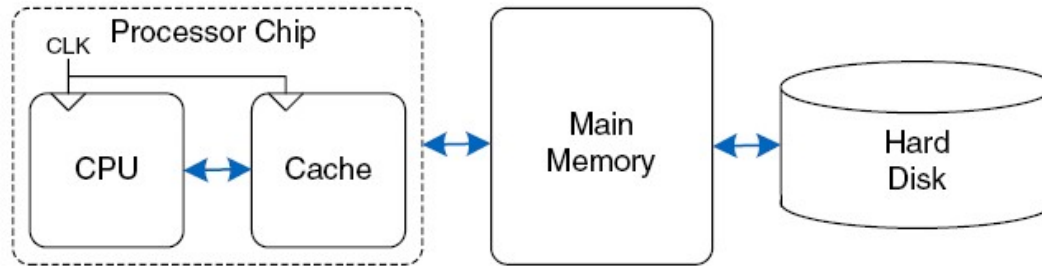
Locality

Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.

- Temporal locality: Recently referenced items are likely to be referenced in the near future.

- Spatial locality: Items with nearby addresses tend to be referenced close together in time.

Locality Example:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data

  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference sum each iteration: **Temporal locality**

- Instructions

  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

Big ideas:

1. Computing / information processing: $y = F(x)$
2. Universality: All Boolean functions can be implemented by wiring a bunch of NAND gates.
3. ALU is programmable (no hard wiring is necessary for a given F)
4. Sequential logic can hold states (memory)
5. Stored programs (von Neumann architecture)
6. Turing complete: Sequence, Branch, and Loop
7. Code reusability and Abstraction: procedures/subroutines
8. Use of stack and recursive calls
9. CPU time = IC x CPI x CC
10. Limitations with data representation: overflow, underflow, 0.1 in binary

Final Exam Info

Four questions
1. Short answers: basic concepts, floating point number representation
2. Assembly programming for dynamic data structures
3. Assembly code in general, assembly-link-load
4. Performance, optimization, cache

Course evaluations

http://www.udel.edu/course-evals/

The evaluation period will end at midnight of December 10, 2021.

Thanks!