CISC 260 Machine Organization and Assembly Language

Spring 2018

Review

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.

CISC260S18 Machine Organization and Assembly Language

#	Week	Topic
1	Feb5	Overview and Data representation
2	Feb12	Boolean logic, gates
3	Feb19	Build a simple computer
4	Feb26	ARM, ISA, Assembly Language
5	Mar5	Assembly programming
6	Mar12	Procedure call, stack
7	Mar19	Assembly programming and Review1
	Midterm March 22	
8	Mar23	Spring break (no classes)
9	Apr2	Floating point
10	Apr9	Assembly programming: Dynamic data structure
11	Apr16	Assembler, Linker, Compiler
12	Apr23	Performance and optimization
13	Apr30	I/O
14	May7	Assembly language programming and security
15	May15	Review2

2018 Spring Semester Final Exams matching ' cisc260'

Course Number	Exam Day	Exam Date	Exam Time	Exam Location
CISC260010	Tuesday	May 22	10:30AM-12:30PM	WHL006

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), MIPS
- Machine language and assembly language
- Stacks and procedure calls
- Memory management
- Performance issues and optimization

As the specific goals of this course, 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, assembly language level
- convert numerical data between different formats
- carry out basic logical and arithmetic operations
- understand memory hierarchy, basic I/O
- understand performance issues and optimization techniques

Sections that are most relevant for the final exam are listed as follows.

Chapter 1.6, 1.10
Chapter 3.1 - 3.5
Chapter 5.1 – 5.4
Appendix A1 – A3, A5, A8
Chapter 02_COD 4e ARM.pdf:
2.1 - 2.10, 2.12 - 2.19

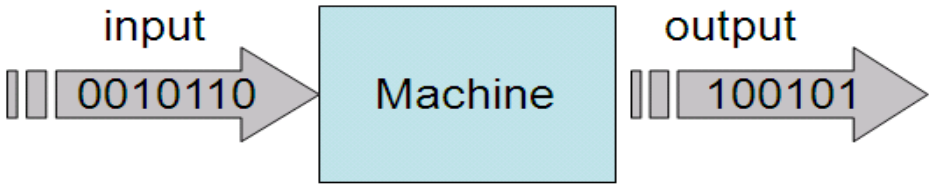
You may also find Appendices A and B are generally useful in supplementing what is covered in the class.

What this course is about?

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

What is computing?

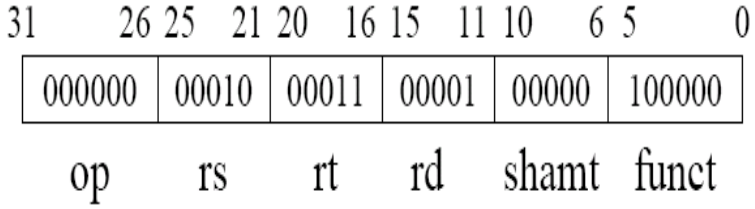
- arithmetic calculating
  e.g., 3 + 2 = 5
- manipulating information
  information? (symbols and interpretation)
  syntax semantics



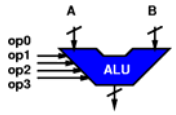
X F( ) Y = F(X)

More layers ...

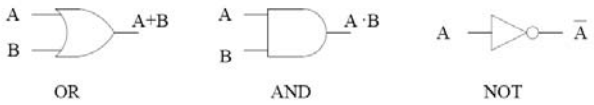
Instruction Set Architecture (ISA):



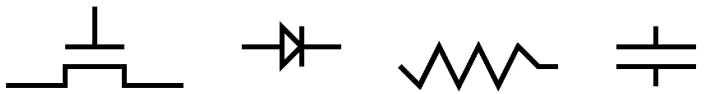
Functional units:



Gates (CPEG 202):



Devices (in silicon)



We will take a bottom-up approach, starting with gates

Control & Data Path

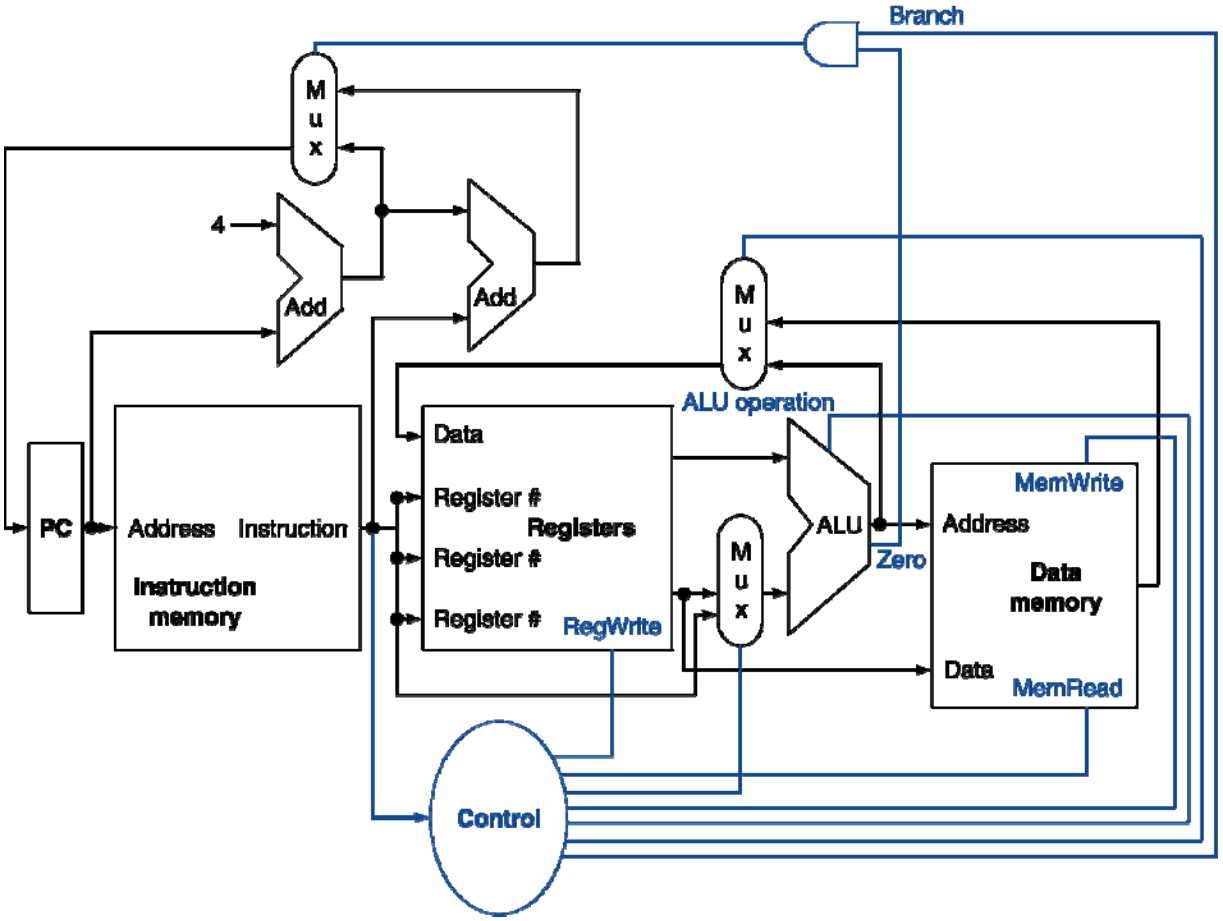


Figure 4.2

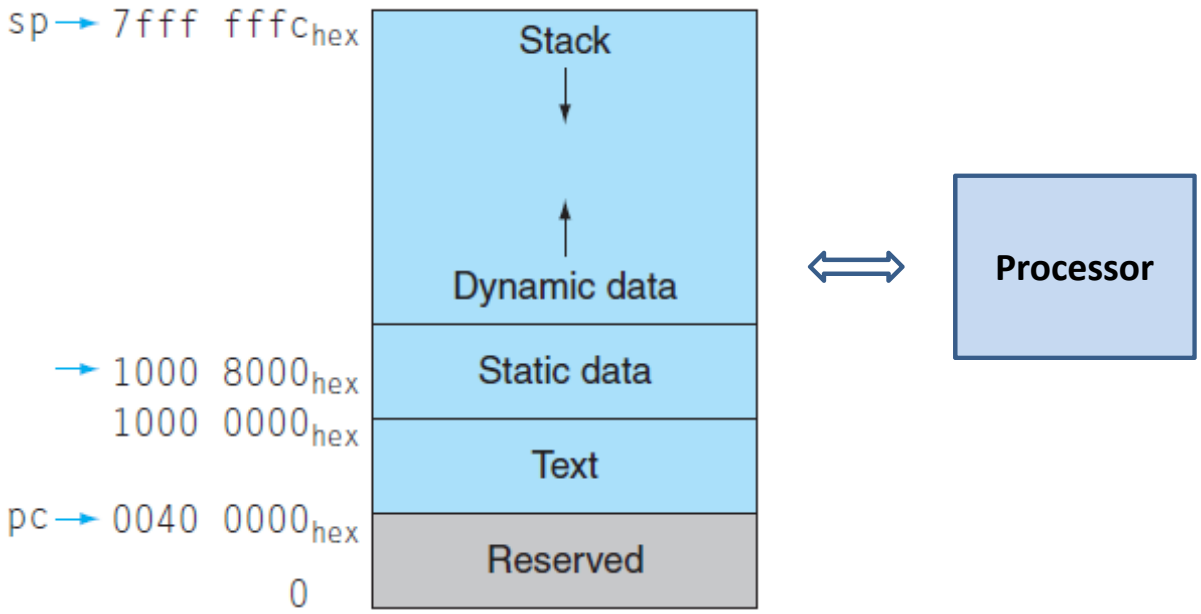


FIGURE 2.13 Typical ARM memory allocation for program and data. These addresses are only a software convention, and not part of the ARM architecture. The stack pointer is initialized to $7fff\ fffc_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at $0040\ 0000_{hex}$. The static data starts at $1000\ 0000_{hex}$. Dynamic data, allocated by `malloc` in C and by `new` in Java, is next. It grows up toward the stack in an area called the heap.

ARM operands

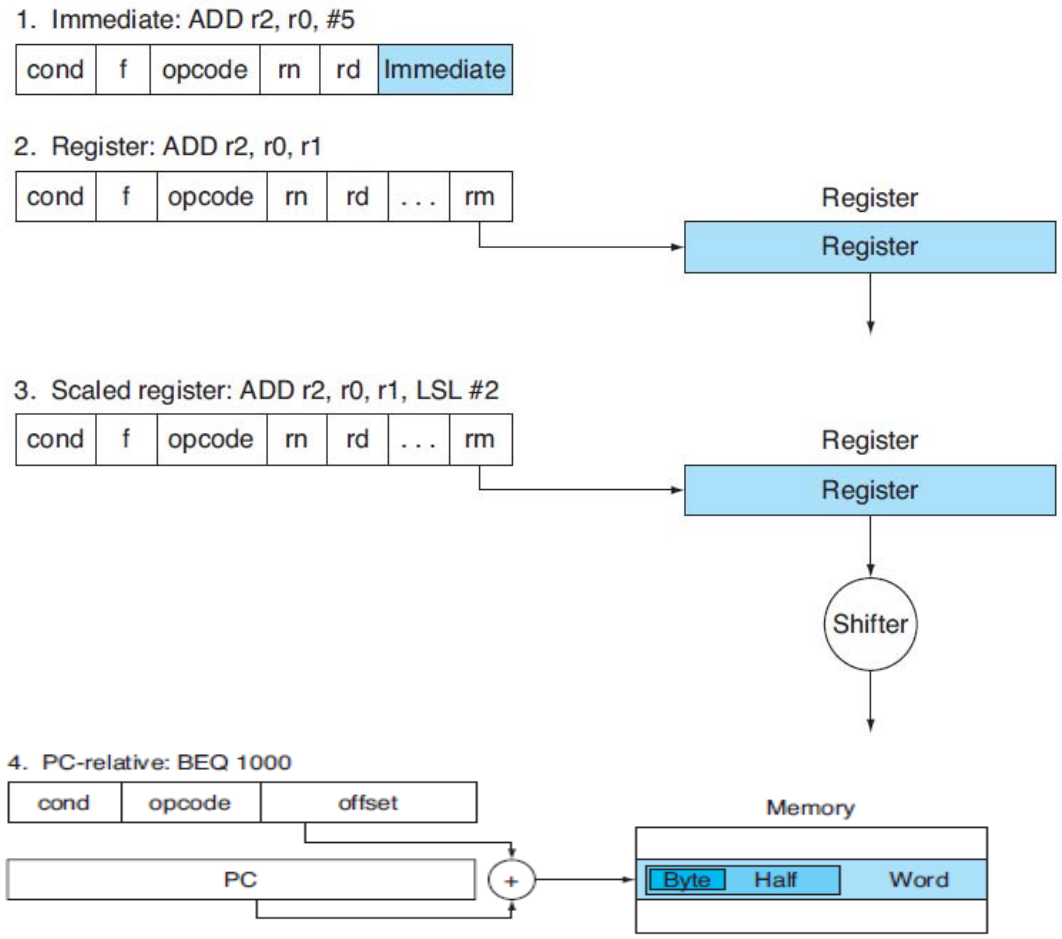
Name	Example	Comments
16 registers	r0, r1, r2,..., r11, r12, sp, lr, pc	Fast locations for data. In ARM, data must be in registers to perform arithmetic, register
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. ARM uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

ARM assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	ADD r1,r2,r3	r1 = r2 – r3	3 register operands
	subtract	SUB r1,r2,r3	r1 = r2 + r3	3 register operands
Data transfer	load register	LDR r1, [r2,#20]	r1 = Memory[r2 + 20]	Word from memory to register
	store register	STR r1, [r2,#20]	Memory[r2 + 20] = r1	Word from memory to register
	load register halfword	LDRH r1, [r2,#20]	r1 = Memory[r2 + 20]	Halfword memory to register
	load register halfword signed	LDRHS r1, [r2,#20]	r1 = Memory[r2 + 20]	Halfword memory to register
	store register halfword	STRH r1, [r2,#20]	Memory[r2 + 20] = r1	Halfword register to memory
	load register byte	LDRB r1, [r2,#20]	r1 = Memory[r2 + 20]	Byte from memory to register
	load register byte signed	LDRBS r1, [r2,#20]	r1 = Memory[r2 + 20]	Byte from memory to register
	store register byte	STRB r1, [r2,#20]	Memory[r2 + 20] = r1	Byte from register to memory
	swap	SWP r1, [r2,#20]	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	Atomic swap register and memory
	mov	MOV r1, r2	r1 = r2	Copy value into register
Logical	and	AND r1, r2, r3	r1 = r2 & r3	Three reg. operands; bit-by-bit AND
	or	ORR r1, r2, r3	r1 = r2 | r3	Three reg. operands; bit-by-bit OR
	not	MVN r1, r2	r1 = ~ r2	Two reg. operands; bit-by-bit NOT
	logical shift left (optional operation)	LSL r1, r2, #10	r1 = r2 << 10	Shift left by constant
	logical shift right (optional operation)	LSR r1, r2, #10	r1 = r2 >> 10	Shift right by constant
Conditional Branch	compare	CMP r1, r2	cond. flag = r1 – r2	Compare for conditional branch
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BEQ 25	if (r1 == r2) go to PC + 8 + 100	Conditional Test; PC-relative
Unconditional Branch	branch (always)	B 2500	go to PC + 8 + 10000	Branch
	branch and link	BL 2500	r14 = PC + 4; go to PC + 8 + 10000	For procedure call

FIGURE 2.1 ARM assembly language revealed in this chapter. This information is also found in Column 1 of the ARM Reference Data Card at the front of this book.

ARM Addressing Modes



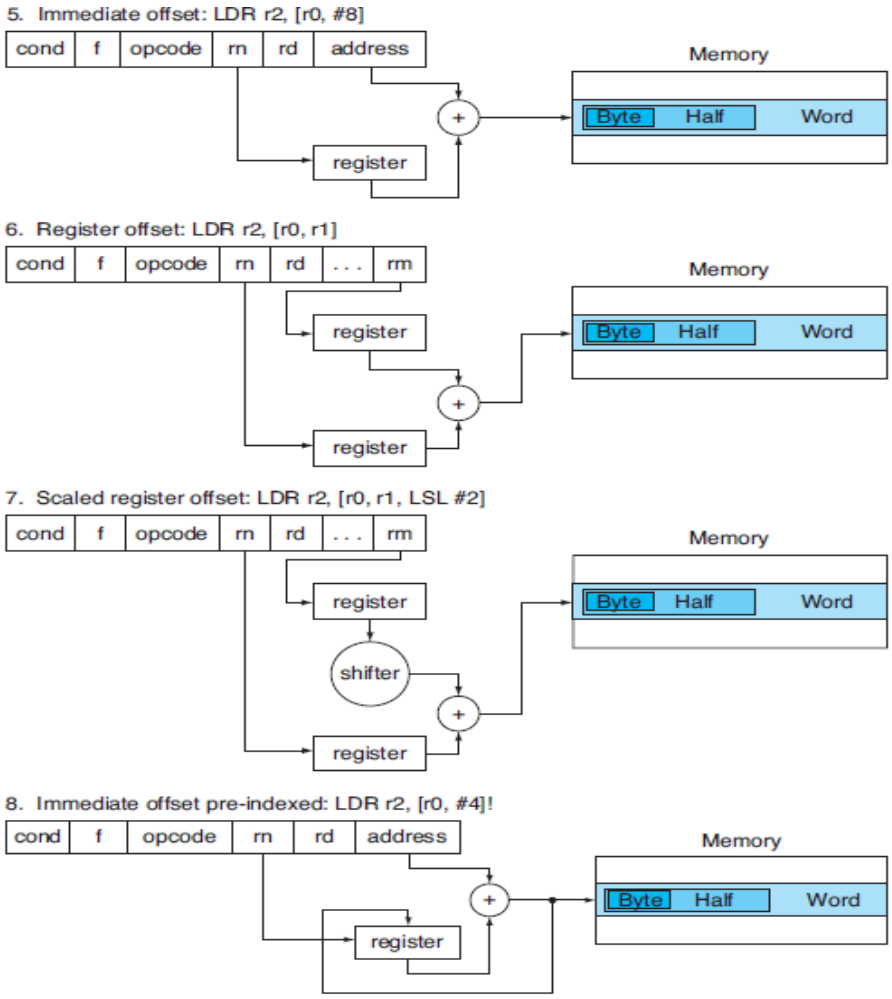
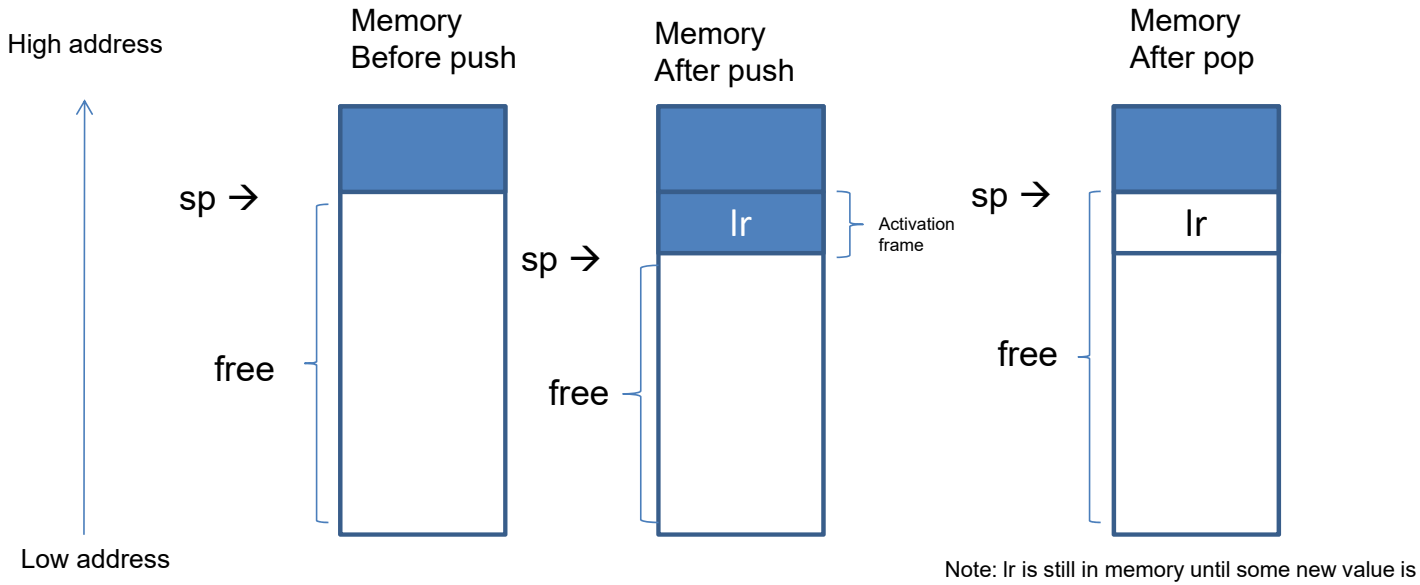


FIGURE 2.17 Illustration of the twelve ARM addressing modes. (*continued*)

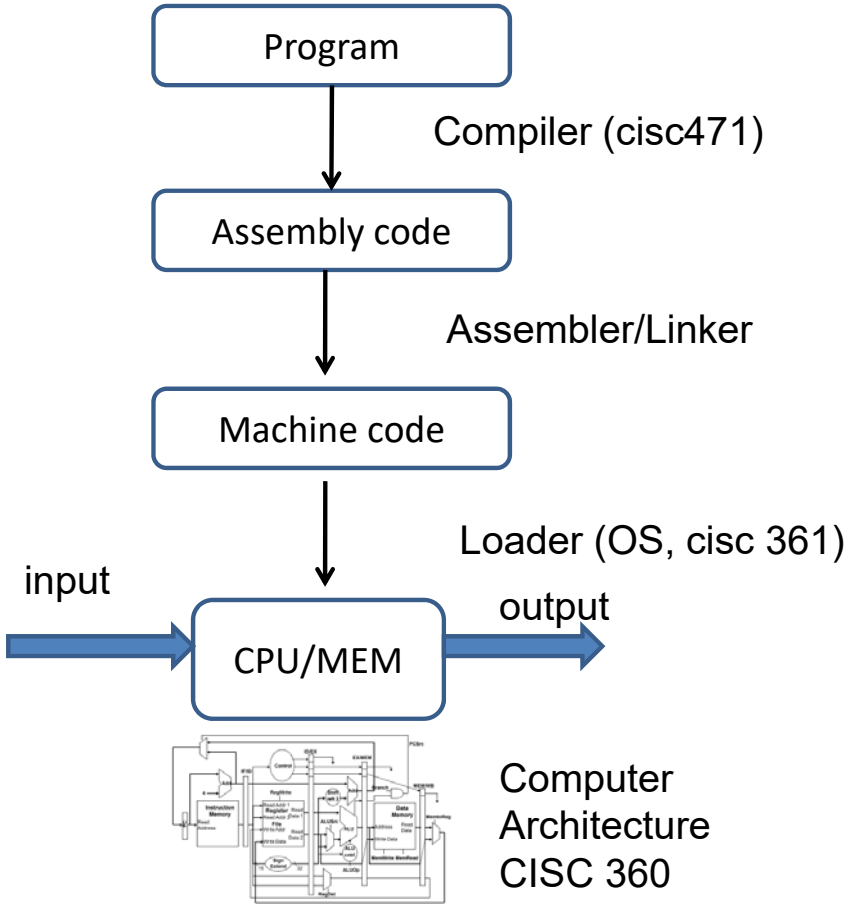
Stack (LIFO)

Acquire storage space, called activation frame

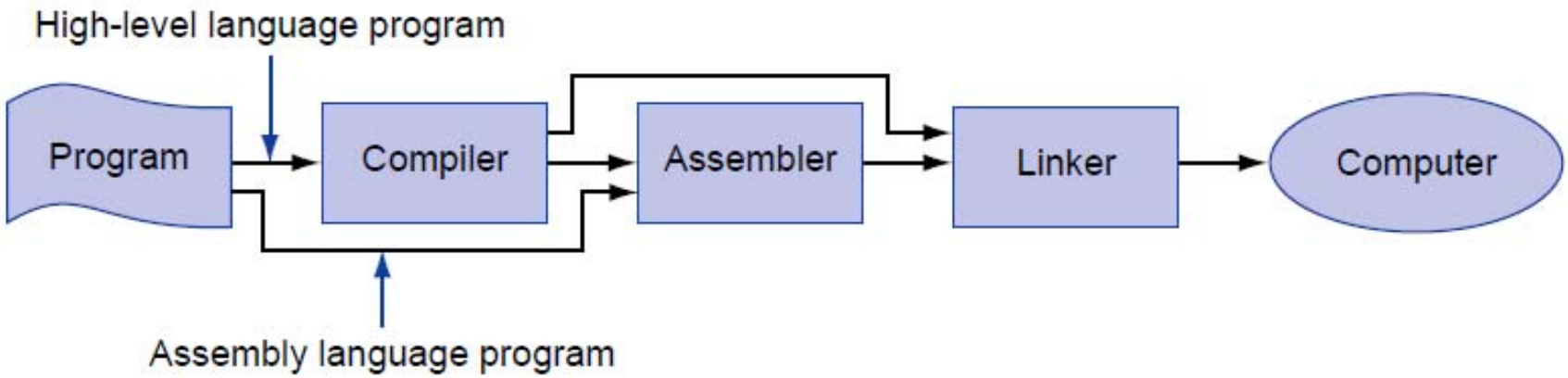
```
SUB     sp, sp, #4          @ push
STR     r14, [sp]           @ push
…
…
…
LDR     r14, [sp]           @ pop
ADD     sp, sp, #4          @ pop
MOV     pc, r14
```

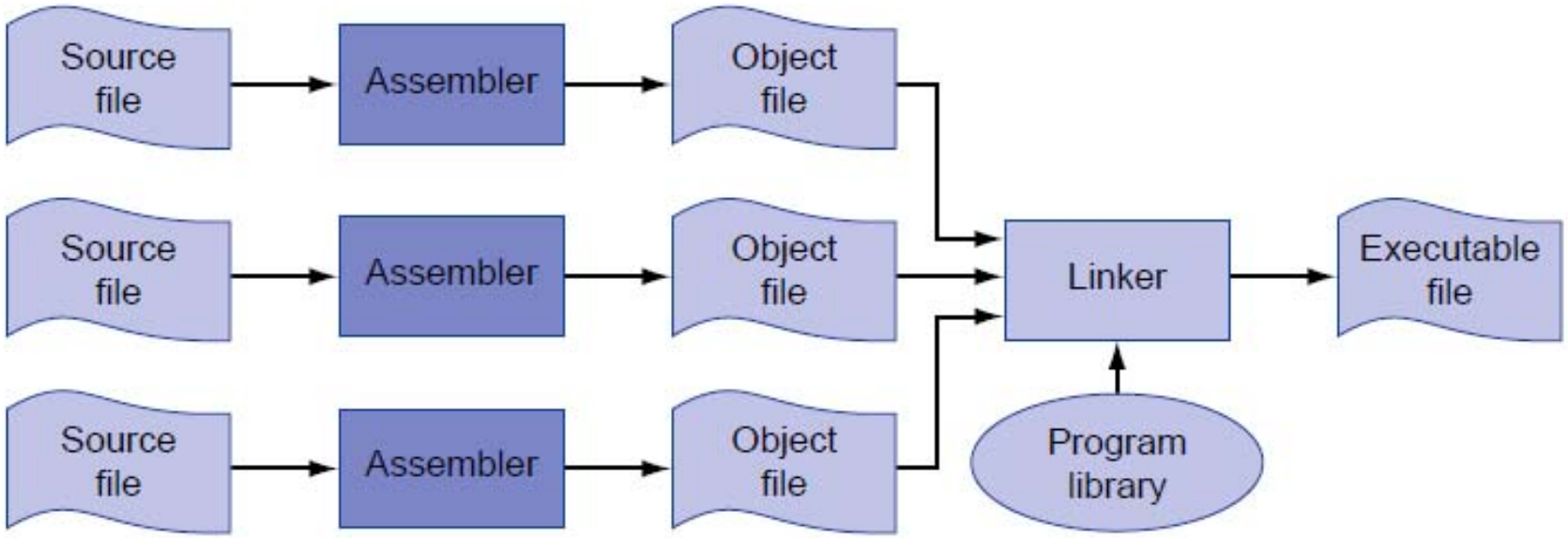


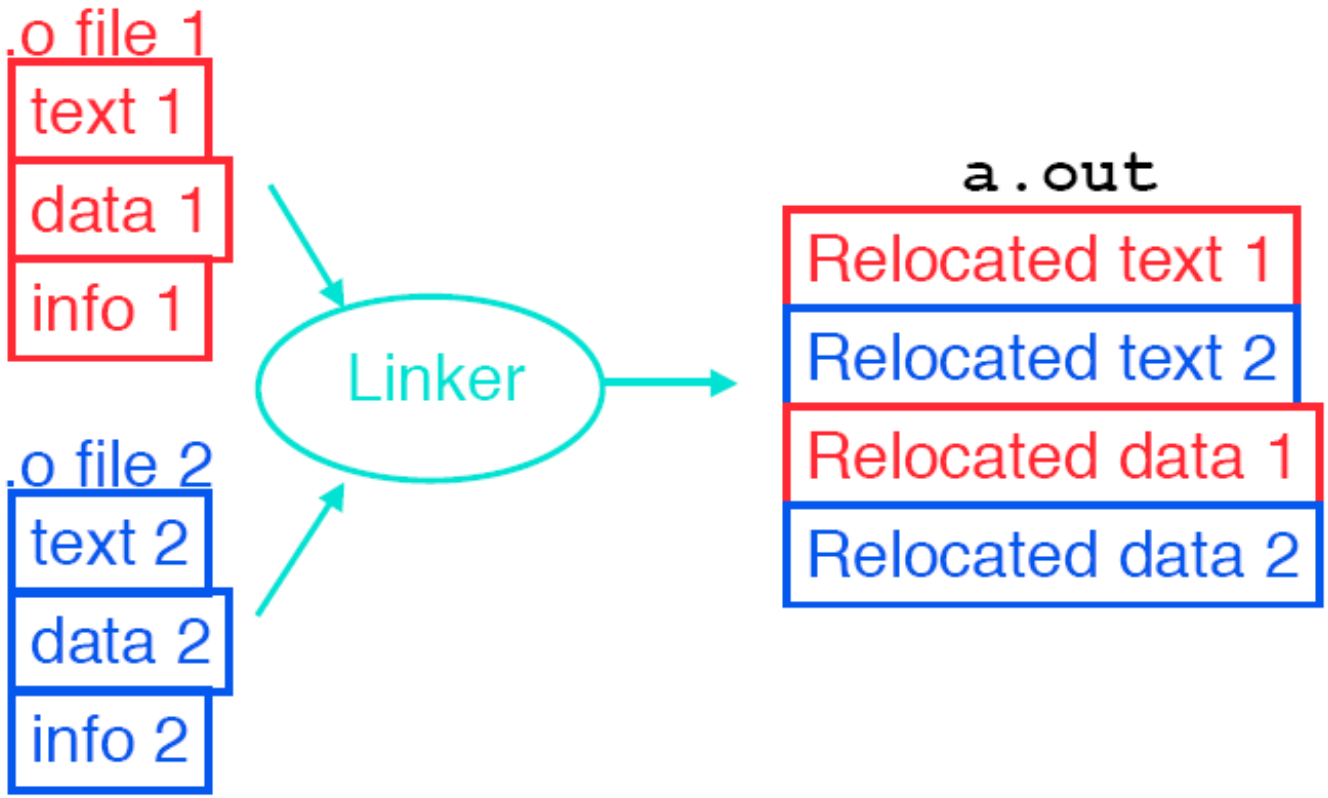
Note: lr is still in memory until some new value is written in the same location. Potential security loophole.



Computer Architecture
CISC 360


High-level language program
Program
Compiler
Assembler
Linker
Computer
Assembly language program


Source file
Assembler
Object file
Source file
Assembler
Object file
Source file
Assembler
Object file
Linker
Executable file
Program library

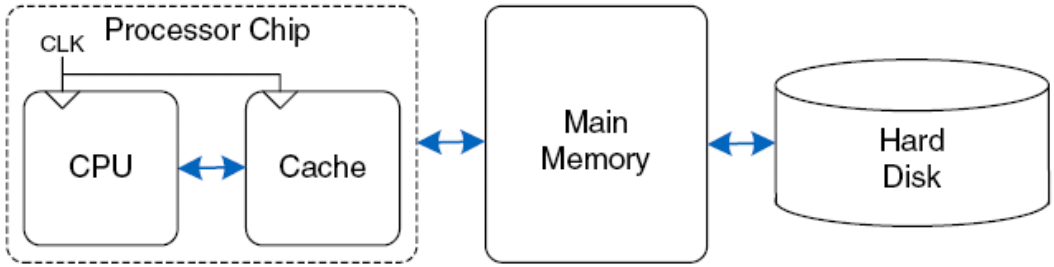

.o file 1
text 1
data 1
info 1
.o file 2
text 2
data 2
info 2
Linker
a.out
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

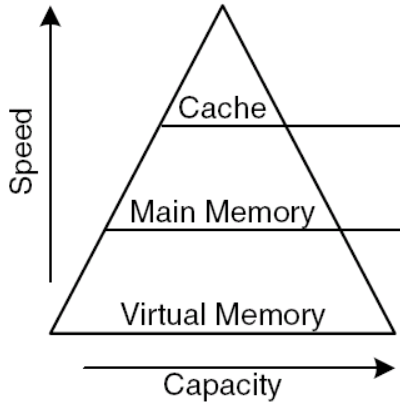
Performance

CPU time = IC x CPI x CC

Amdahl's Law:

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





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

Course evaluations

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

The evaluation period will end at midnight of Sunday, May 16, 2018.

Thanks!