

# CISC 260 Machine Organization and Assembly Language

Spring 2019

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.

## CISC260 Tentative Schedule (S19)

#	Week	Topic
1	Feb11	Overview and Data representation
2	Feb18	Boolean logic, gates
3	Feb25	Build a simple computer
4	Mar4	ARM, ISA, Assembly Language
5	Mar11	Assembly programming
6	Mar18	Procedure call, stack
7	Mar25	Assembly programming and Review1
Midterm March 28		
8	Apr1	Spring break (no classes)
9	Apr8	Floating point
10	Apr15	Assembly programming: Dynamic data structure
11	Apr22	Assembler, Linker, Compiler
12	Apr29	Performance and optimization
13	May6	I/O and more assembly programming
14	May13	Assembly language programming and security
15	May16	Review2

## 2019 Spring Semester Final Exams matching 'cisc260'

If you have questions, contact the Office of the Registrar by writing to [schedoffice@udel.edu](mailto: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 
CISC260010	Thursday	May 30	3:30 PM-5:30 PM	KRB004 

The final exam will be open-note, like the midterm exam.

# **Reading:**

Chapter 1.1 - 1.5

Chapter 2.1 - 2.4, 2.8;

Chap 3.1; 3.2.1 - 3.2.4;

Chap 5.1 - 5.3

Chap 6.1 - 6.9

Chap 8.1 – 8.3.2

Chap 9.1 – 9.2

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/subroutines
8. Stack and recursive calls

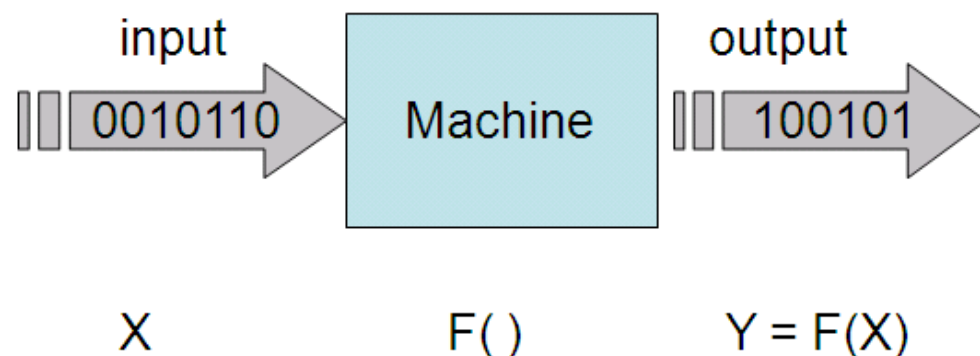


What this course is about?

- 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



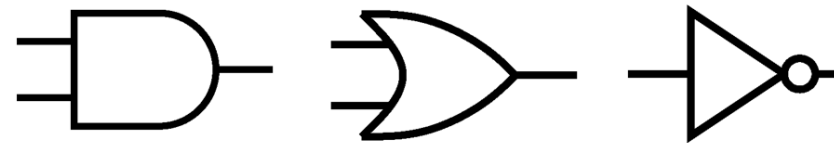
# Any function can be implemented in Boolean logic

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

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

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

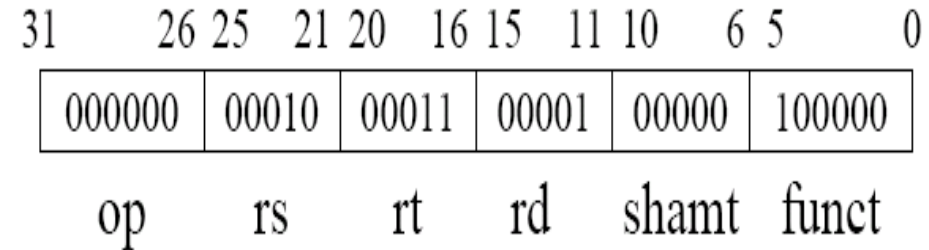
$X = \sim A \& B \& C \mid A \& \sim B \& \sim C \mid A \& B \& \sim C$   
 $Y = \sim A \& \sim B \& \sim C \mid A \& B \& \sim C$



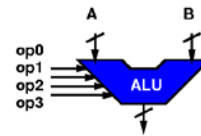
# Build a computer

More layers ...

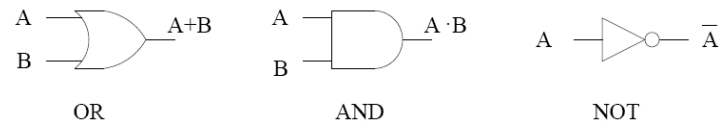
Instruction Set Architecture  
(ISA):



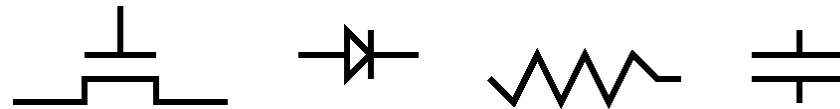
Functional units:



Gates (CPEG 202):

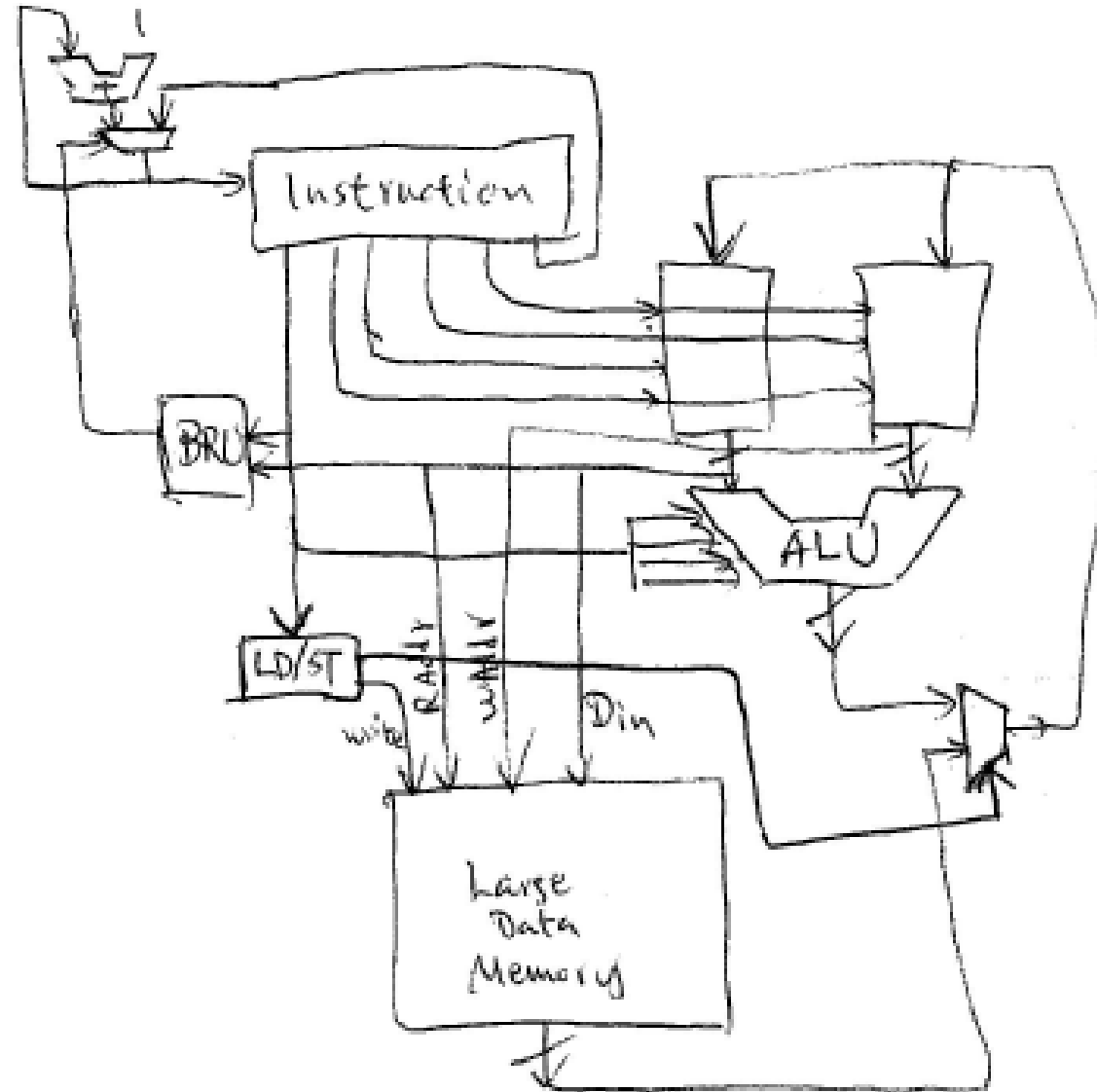


Devices (in silicon)



We take a bottom-up approach, starting with gates

# A simple computer



# ARM (32bit, single cycle computer)

- Data path
- Parse/Decoder
- Control/Mux

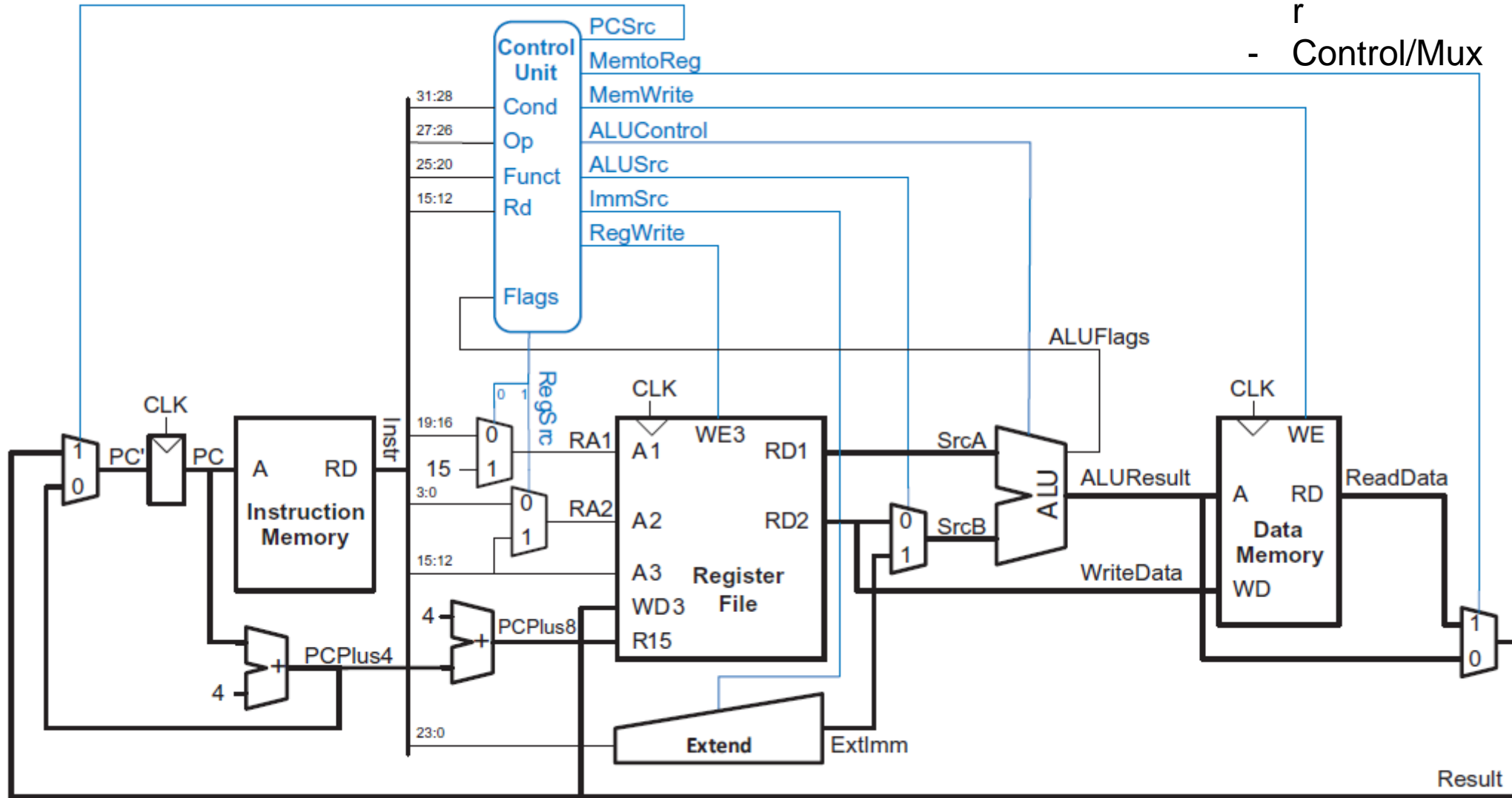


Figure 7.13 Complete single-cycle processor

# A Programmer's Perspective

## Memory

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

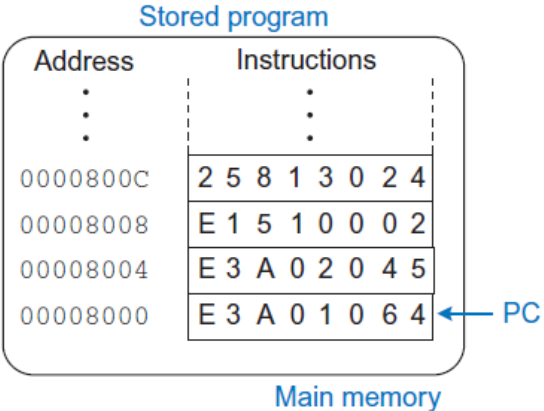


Figure 6.28 Stored program

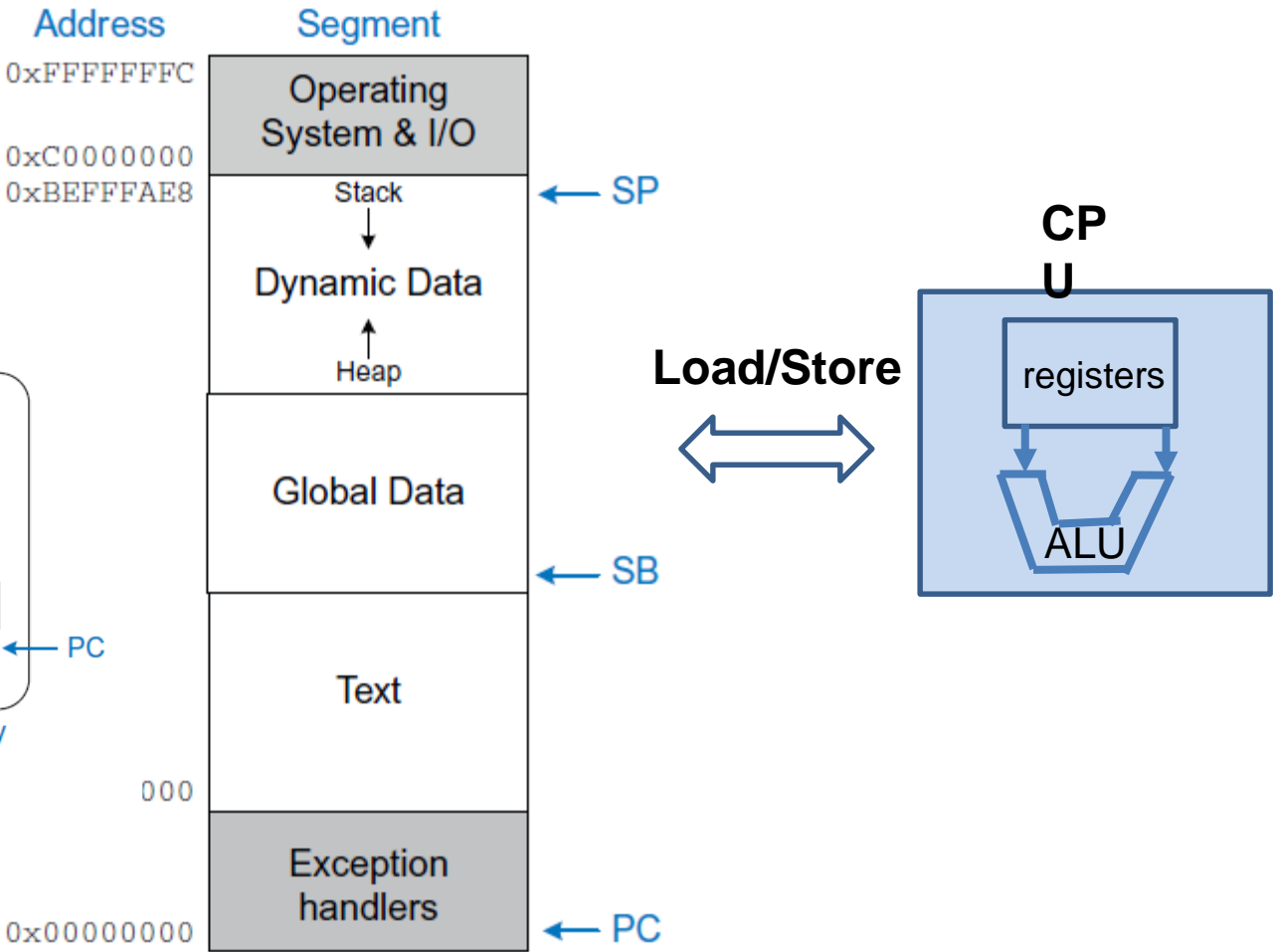


Figure 6.30 Example ARM memory map

# Basic ARM Instructions

## Data processing:

ADD r1, r2, r3

## Meaning

$r1 \leftarrow r2 + r3$

SUB r1, r2, r3

$r1 \leftarrow r2 - r3$

AND r1, r2, r3

$r1 \leftarrow r2 \& r3$

ORR r1, r2, r3

$r1 \leftarrow r2 | r3$

MVN r1, r2

$r1 \leftarrow \sim r2$

LSL r1, r2, #10

$r1 \leftarrow r2 \ll 10$

LSR r1, r2, #10

$r1 \leftarrow r2 \gg 10$

## Data Transfer

### (Memory):

LDR r1, [r2, #20]

$r1 \leftarrow \text{Memory}[r2 + 20]$

STR r1, [r2, #20]

$\text{Memory}[r2 + 20] \leftarrow r1$

SWAP r1, [r2, #20]

$r1 \leftrightarrow \text{Memory}[r2 + 20]$

MOV r1, r2

$r1 \leftarrow r2$

## Branching:

CMP r1, r2

Condition flag NZCV is set per the outcome of r1-r2

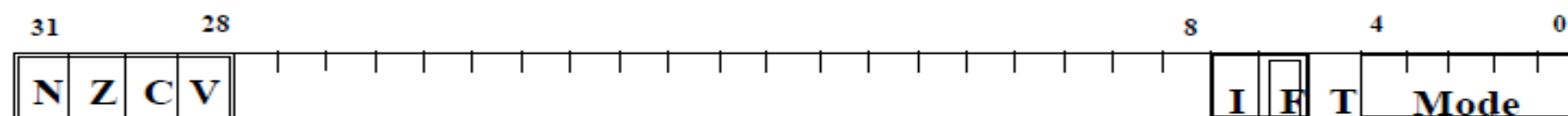
B label

$\text{PC} \leftarrow \text{Address of label}$

BL label

$\text{PC} \leftarrow \text{Address of label}, r14 \leftarrow \text{PC} + 4$

# The Program Status Registers (CPSR and SPSRs)



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**verflowed

- \* **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)

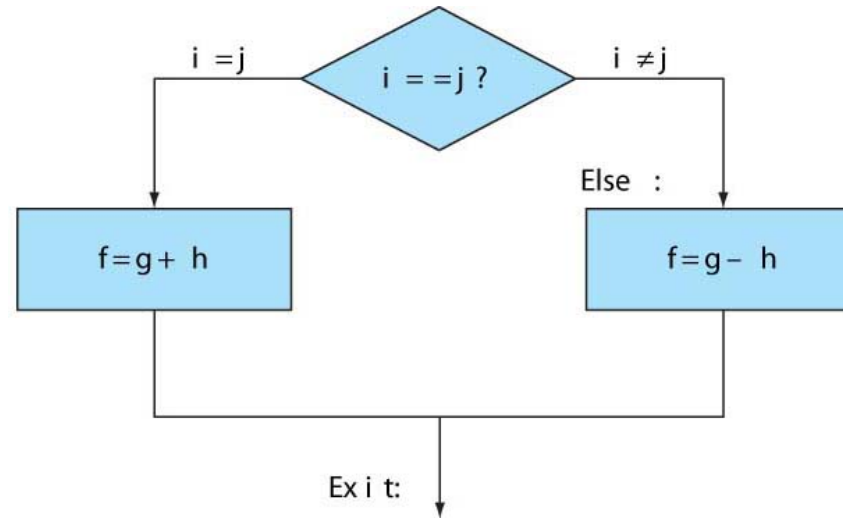
♠ “If-then-else”

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)
```

# 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**)



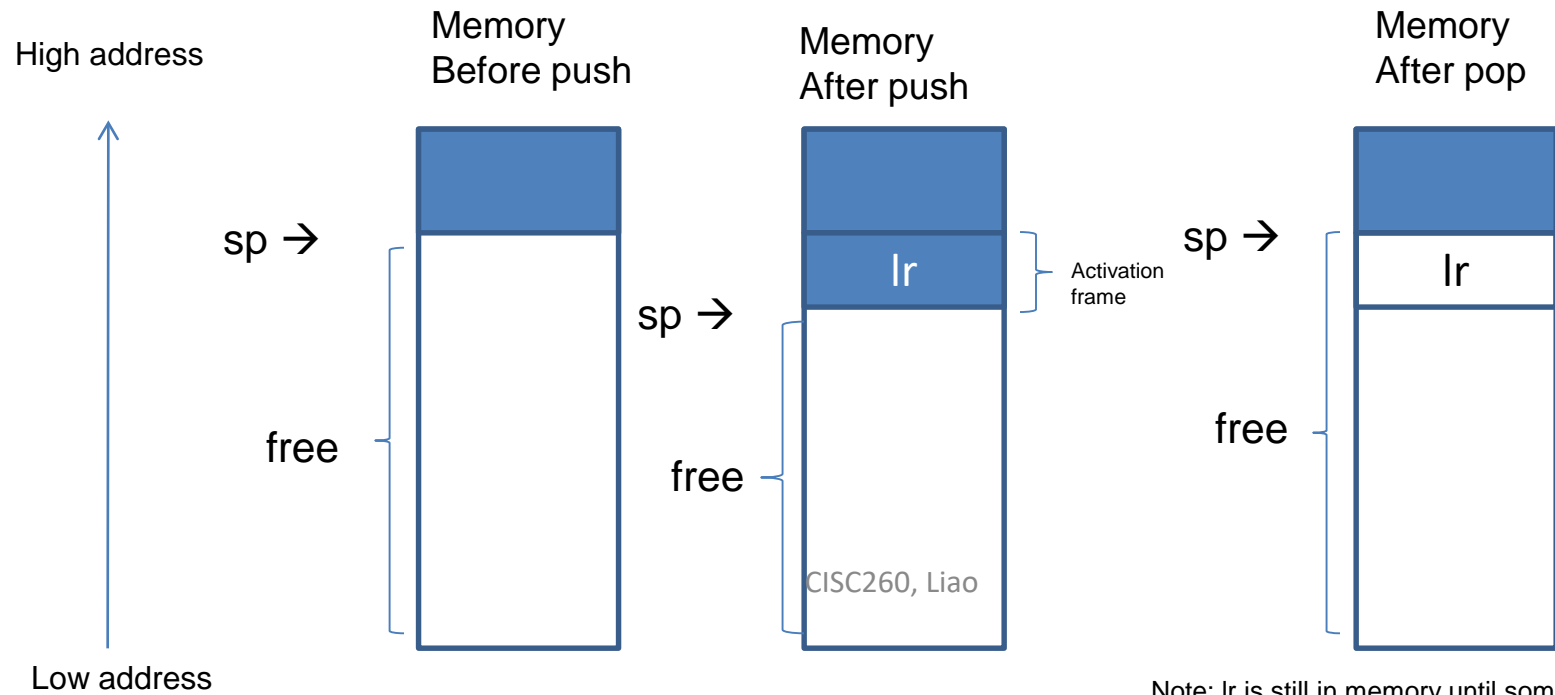
callee

ARM conventions:

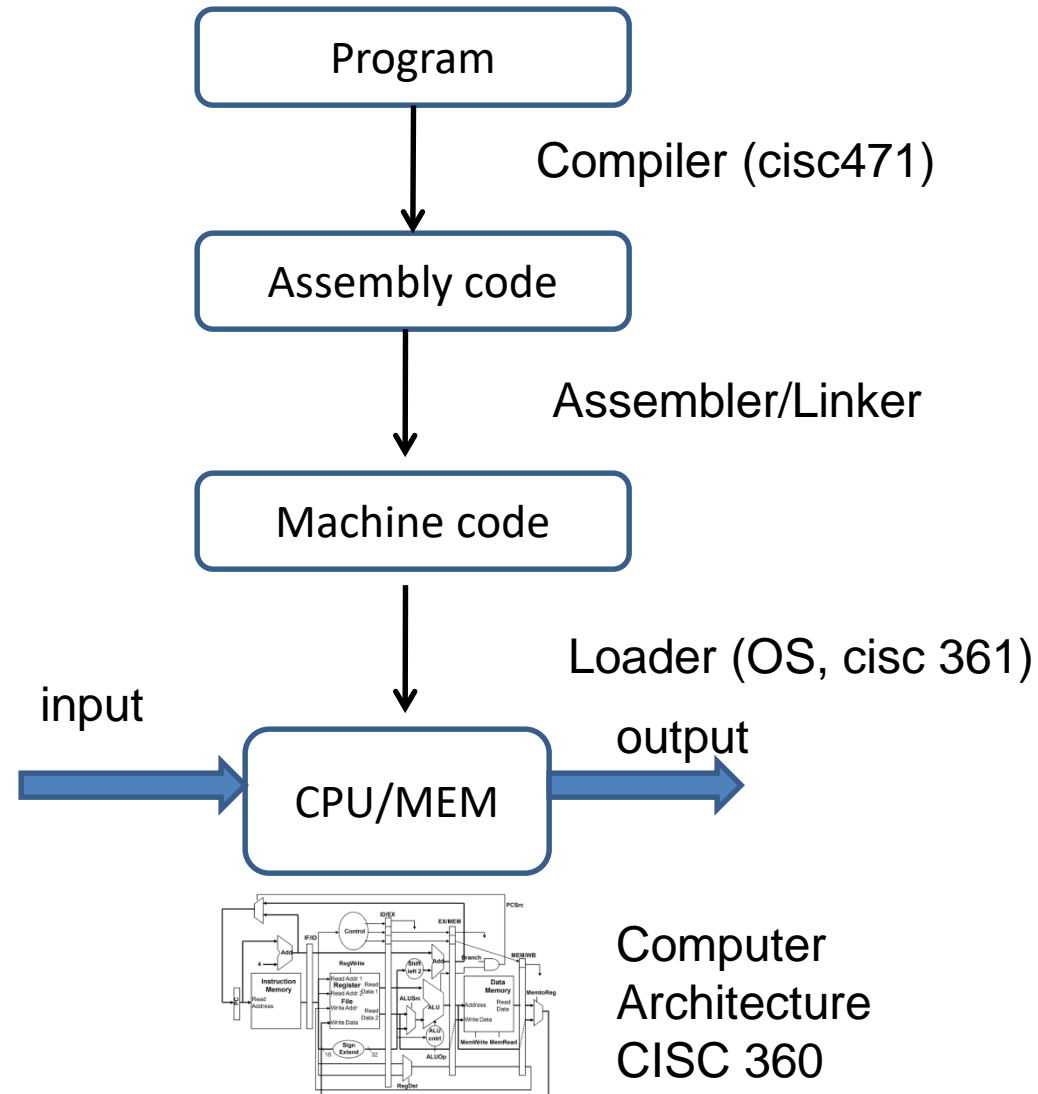
- r0 - r3, r12: registers for storing arguments or scratch registers to used by the callee (not preserved).
- r4-r11: registers that need to be preserved, if used by callee.
- lr: register storing the return address (r14)
- sp: stack pointer (r13)

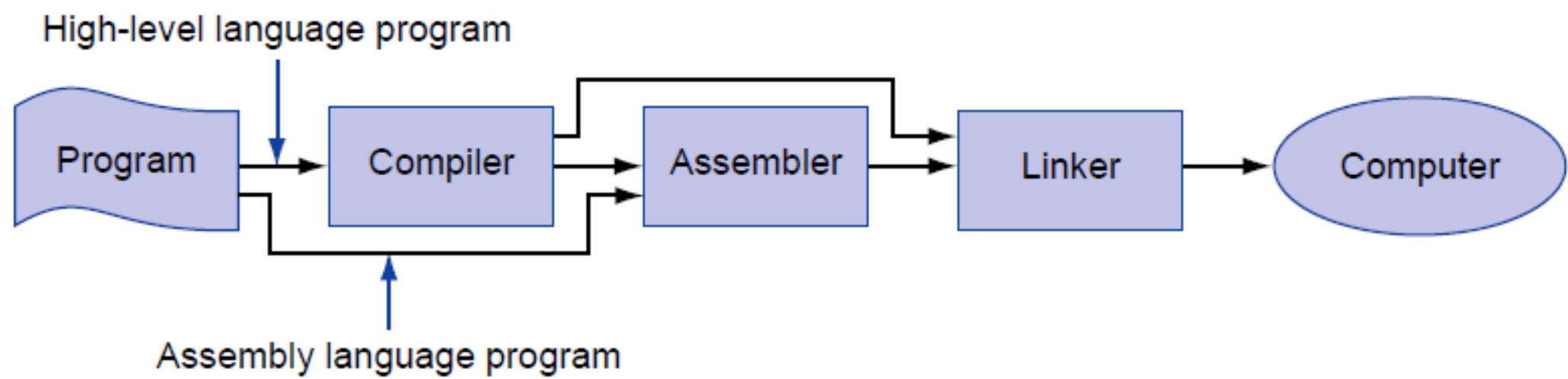
***Subroutine: use stack to maintain activation frames for subroutine calls***

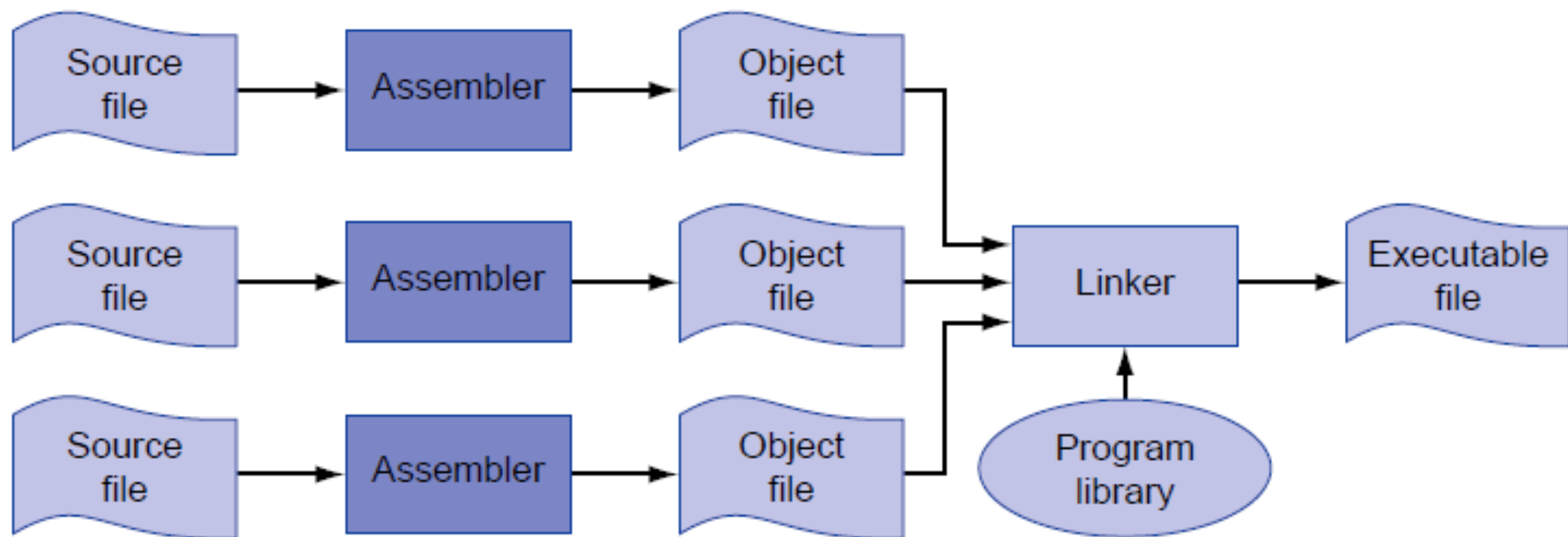
SUB	sp, sp, #4	@ push
STR	r14, [sp]	@ push
...		
...		
...		
LDR	r14, [sp]	@ pop
ADD	sp, sp, #4	@ pop
MOV	pc, r14	

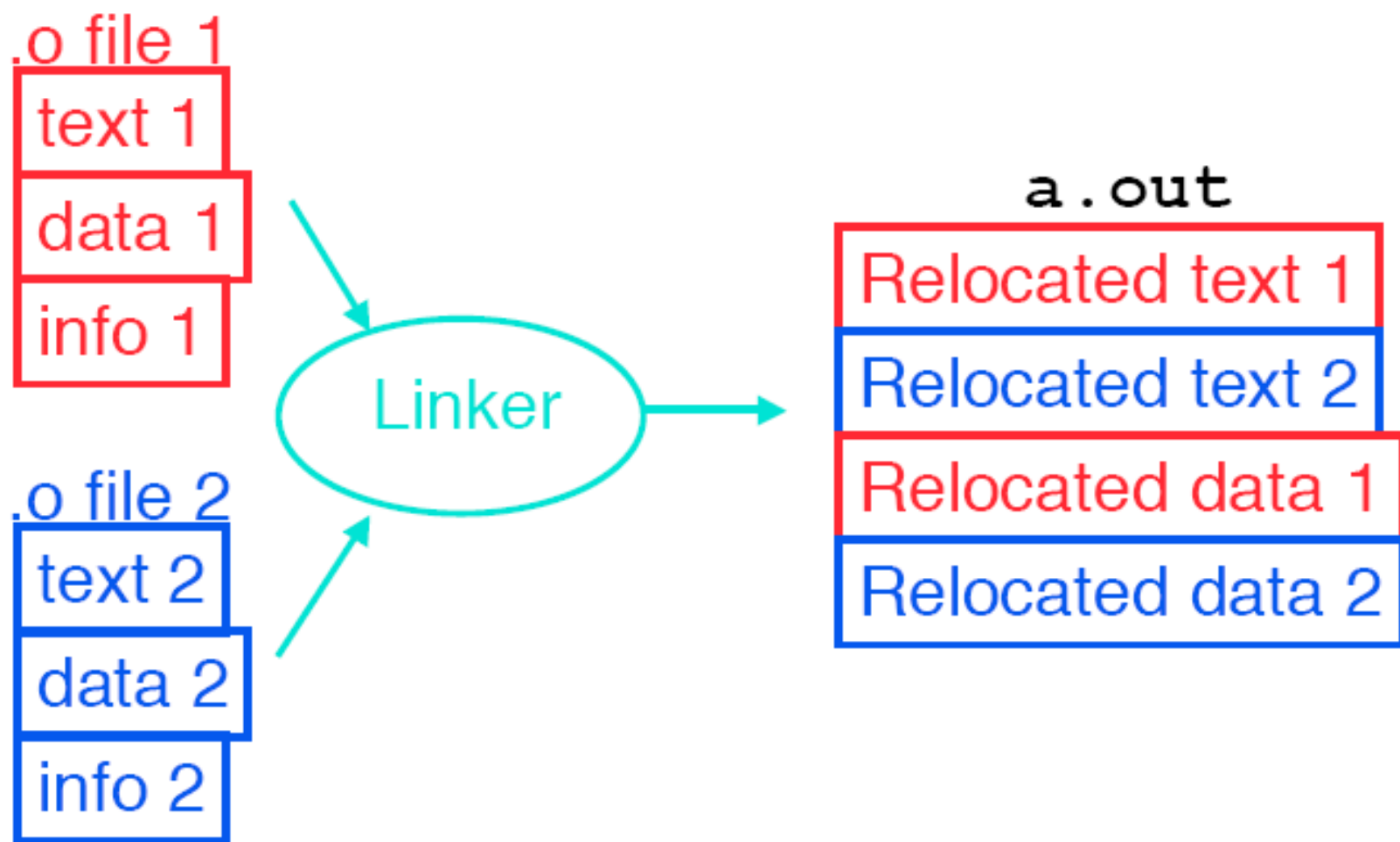


Note: lr is still in memory until some new value is written in the same location. Potential security loophole.









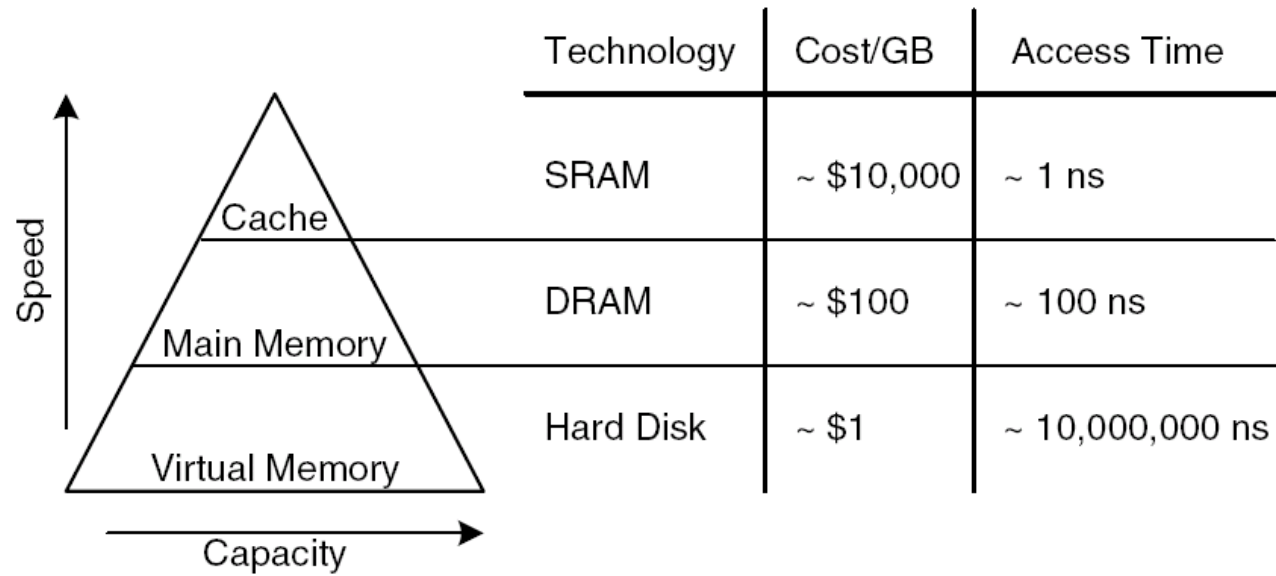
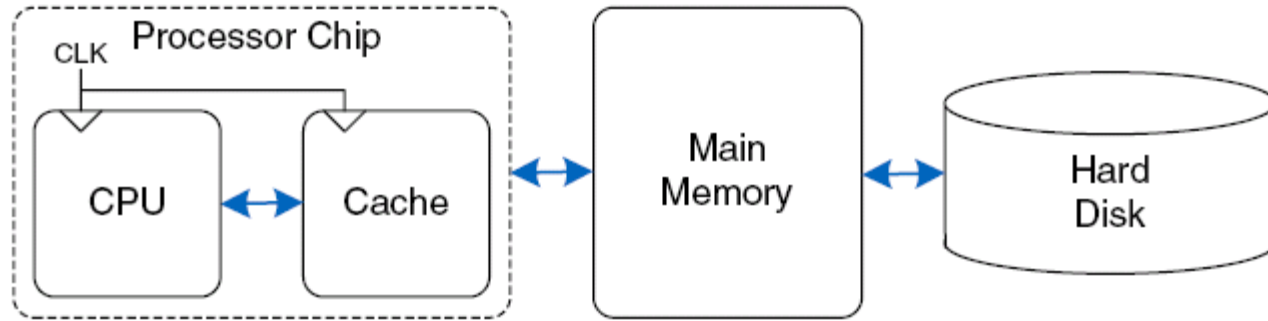


# Performance

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{CC}$$

Amdahl's Law:

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



# 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:

- 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**

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

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 May 21, 2019.

**Thanks!**