CISC 260 Machine Organization and Assembly Language

Spring 2018

Midterm Review

CISC260 Machine Organization and Assembly Language
Tentative schedule

#	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 (Nov.8 Election Day. no class)
12	Apr23	Performance and optimization
13	Apr30	I/O
14	May7	Assembly language programming and security
15	May15	Review2

Reading:

Chapter 3.1 - 3.4
Appendix A1 – A3, A5, A8
Chapter 02_COD 4e
ARM.pdf: 2.1 - 2.8, 2.10

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

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, are implemented at the machine and assembly language level
- convert numerical data between different formats
- carry out basic logical and arithmetic operations

Big ideas:

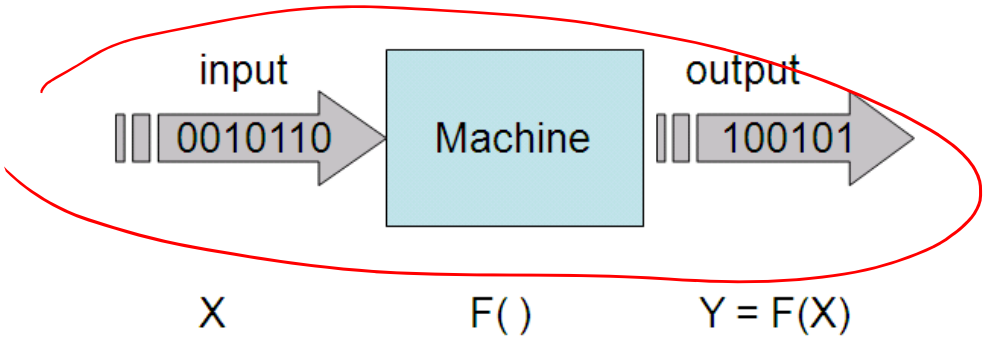
1. Computing / information processing: y = F(x)
2. Universality: All boolean functions can be implemented by wiring a bunch of NAND gates.
3. ALU is programmable (no hard wiring is necessary for a given F)
4. Sequential logic can hold states (memory)
5. Stored programs (von Neumann architecture)
6. Turing 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



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

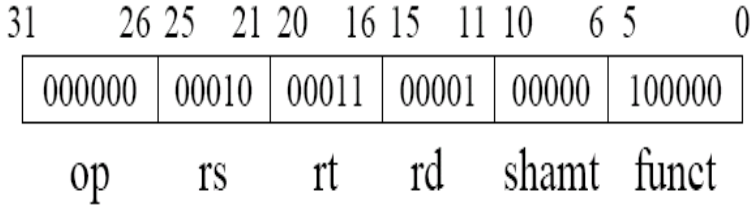
X=~A&B&C | A&~B&~C | A&B&~C

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

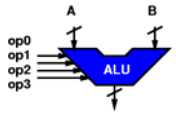
Build a computer

More layers ...

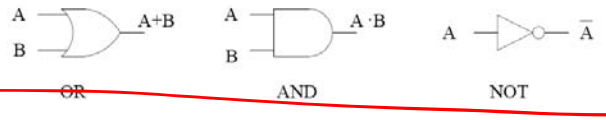
Instruction Set Architecture (ISA):



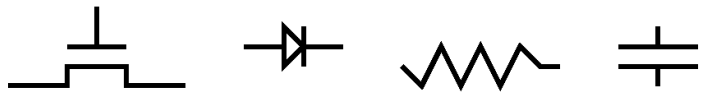
Functional units:



Gates (CPEG 202):

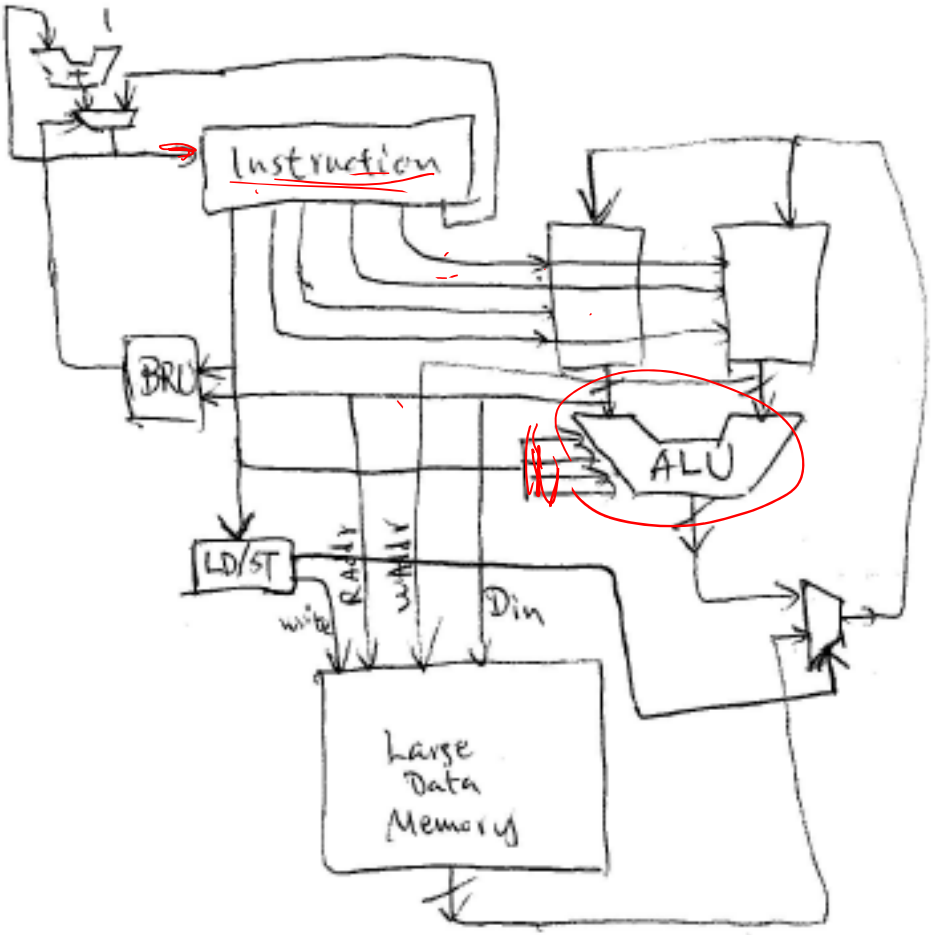


Devices (in silicon)

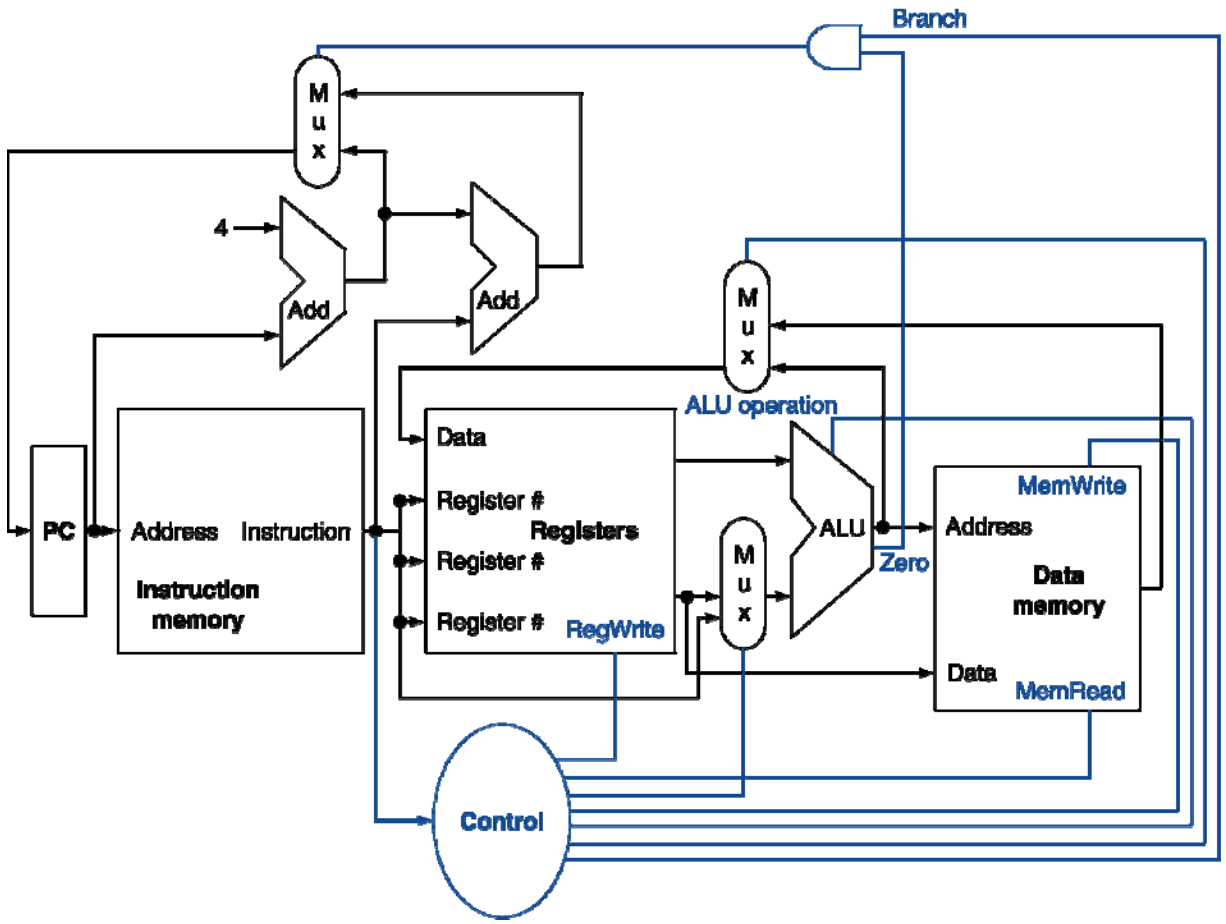


We take a bottom-up approach, starting with gates

A simple computer



ARM Datapath With Control



Note: The following is for the 32-bit ARM7, see Chapter 02_COD 4e ARM

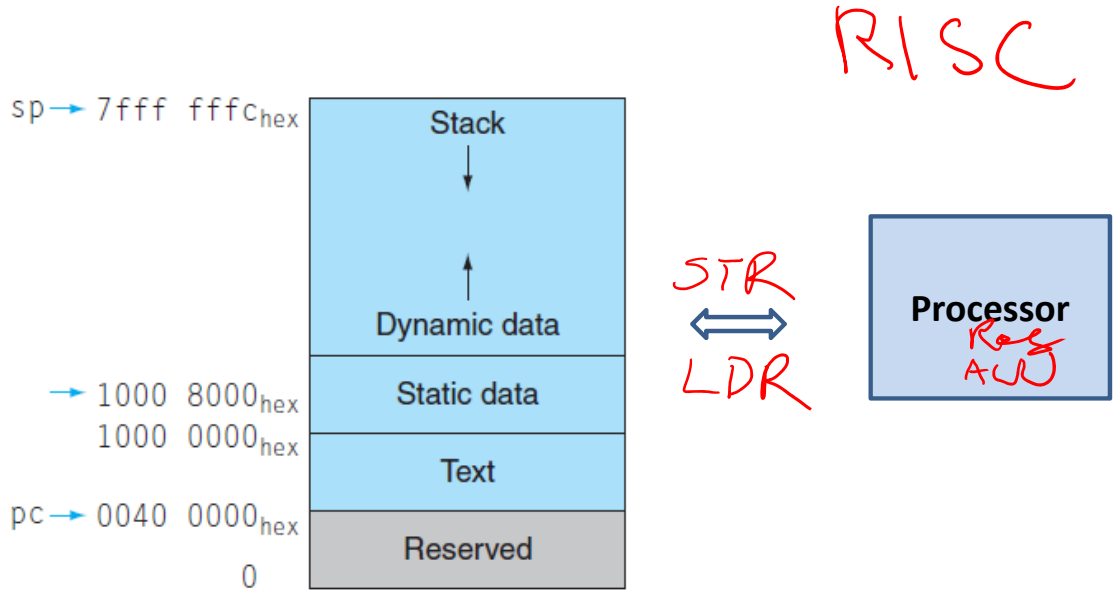


FIGURE 2.13 Typical ARM memory allocation for program and data. These addresses are only a software convention, and not part of the ARM architecture. The stack pointer is initialized to 7fff fffc$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0040 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.

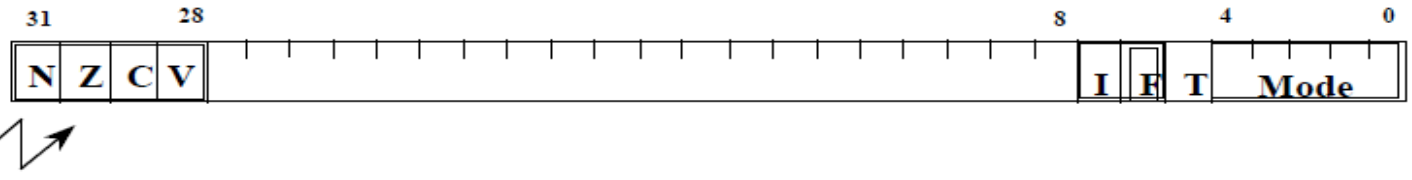
ARM assembly language

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.

More instructions: MUL, BNE, BLE, BLT, BGE, BGT, BLO

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**V**erflowed

* **Mode Bits**
  M[4:0] define the processor mode.

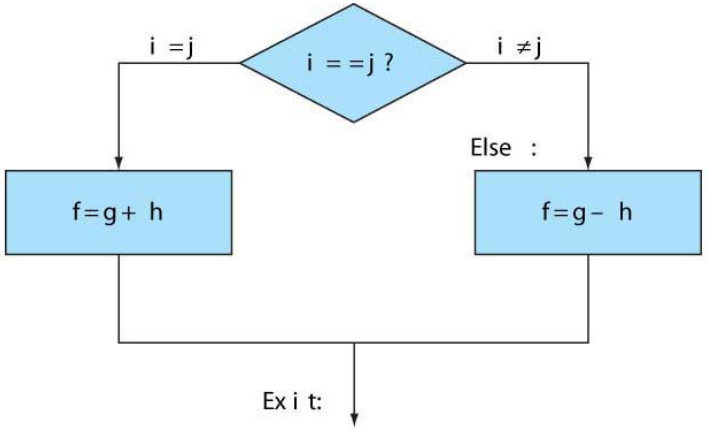
* **Interrupt Disable bits.**
  I = 1, disables the IRQ.
  F = 1, disables the FIQ.

* **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)
```

Loop

Example:

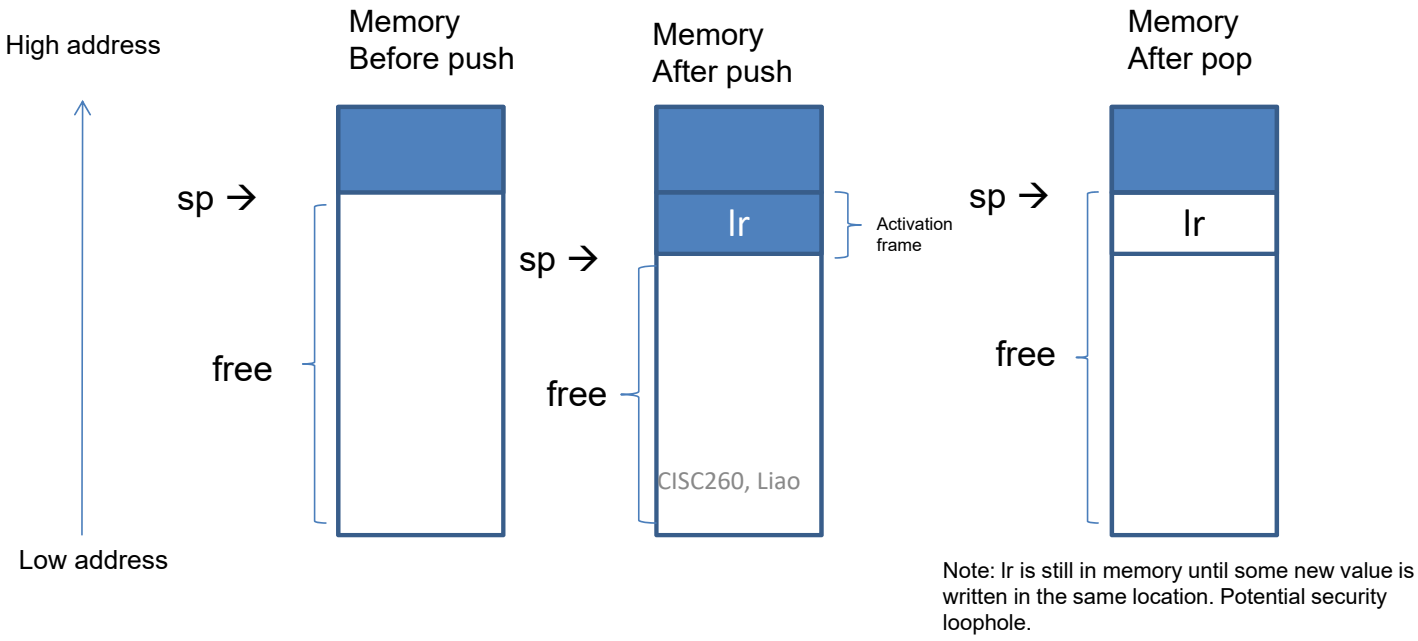
```
        while (save[i] == k)
                i += 1;
```

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:
```

➢ ***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.