Unit 16

Computer Organization

Design of a Simple Processor

INSTRUCTION SET OVERVIEW

Instruction Set Overview

- The instruction set defines the ___________ interface to the ___________ and memory system
- Most processors define their own ________ instruction set unless they are trying to be compatible with some other vendor
  - Which means code compiled for one processor will ______________ on another
- Instruction set is the ___________ the HW processor can understand and the SW is composed with
  - Usually the compiler is the one that translates the high level software into the 1s and 0s (aka ______________) that control the processor

Components of the Instruction Set

- Instruction sets specify…
  - Maximum bit widths for ______ and __________
    - 8-, 16-, 32- or 64-bit
  - Which instructions are implemented
    - ADD, NEGate, SUB, MUL
  - How many ___________ the processor provides to instructions
    - Usually 16 to 64
  - How each instruction is __________________ binary (aka "machine code")

Instruction Set Architecture (ISA)

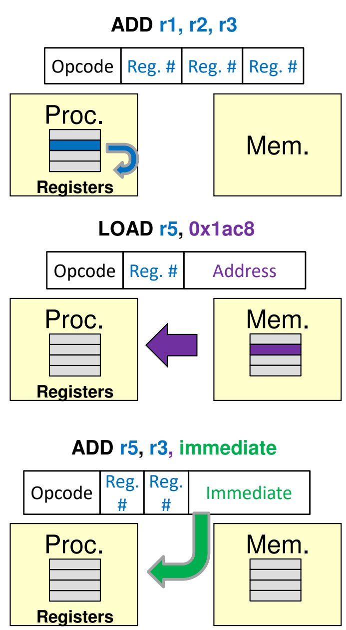
- 2 instruction set approaches
  - ________ = __________ instruction set computer
    - Large, rich set of instructions (vocabulary)
    - Useful when programmers actually wrote ______________
    - More work per instruction, slower clock cycle
  - ________ = __________ instruction set computer
    - Small, basic, but *sufficient* instruction set (vocabulary)
    - With maturation of ______________, manual programming at assembly level is more rare
    - Less work per instruction, faster clock cycle
    - Usually a simple and small set of instructions with regular format facilitates building faster processors

Kinds of Instructions

- Most assembly/machine instructions fall into one of three categories
- ________________________
  - Example: `ADD r1, r2, r3` // r1 = r2 + r3
- **_________________ (to and from memory)**
  - Example: `LOAD r1, addr` // set register 1 to the value in mem. at addr
- _________________
  - Example: `JUMP addr` // Go to instruction in mem. at addr
- Notice that each instruction has predefined interpretation of the operands
  - `LOAD r1, addr` // interprets 1st operand as a register number and 2nd as an address
  - The interpretation of the meaning of the operand is part of the instruction set and known as "addressing modes"

Operands

- Addressing modes refers to how an instruction specifies ***where*** the operands are
- Can be in a
  - ______________,
  - ___________________, or a
  - ___________ that is part of the instruction itself (**aka. ________________ value**)
- Most RISC processors: All data operands for arithmetic/logic instructions must be in a register
  - This allows the hardware to be simpler and faster



DESIGN OF A SIMPLE INSTRUCTIONS SET AND PROCESSOR

What Shall We Do?

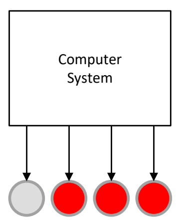
- Let's design a simple processor to understand the entire flow from writing software to designing the hardware
  - This may not be the most advanced processor but the goal is to give you a fully working example from software to hardware

The Instruction Set (1)

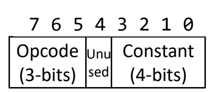
- To start we will define the instruction set
- Let's use __-bit data values (i.e. all data operands will be __-bits)
- Let's make this a simple calculator-like processor that can perform at least the following 3 operations:
  - ____
  - ____
  - ____
- Goal is to evaluate simple arithmetic expressions: (7+4-5)&3
- To keep the number of bits needed to code an instruction to a minimum, let's use an ______________________ architecture where the _____ register is always one ________________
  - ADD 7 means: ___________
  - SUB 5 means: ___________

The Instruction Set (2)

- Let's assume the output of this computer is just 4 LED's to display a 4-bit binary number
- We'll provide some additional instructions to help us perform the calculations:
  - ____________________
  - ____________________
  - ____________________
- That leaves us with 6 total instructions
  - How many bits do we need for the opcode of our instructions? __________
- If we want to store data/constants in our instructions (e.g. ADD 7, SUB 5) how many additional bits do we need in our instruction? ____________
- Instructions need __ opcode + ___ data bits = __-bits
  - Let's round up to 8-bits for each instruction



Output LEDs (Display = 7 = 0111_2)



Chosen Instruction Format

Compilation

- Consider the following "high-level" code
  `(7 - 4 + 6) & 3`
- "Compile" it to an appropriate instruction sequence (i.e. assembly)
  - Assembly refers to the human readable syntax of each instruction
- Now we need to convert to binary…

Instruction Set Summary

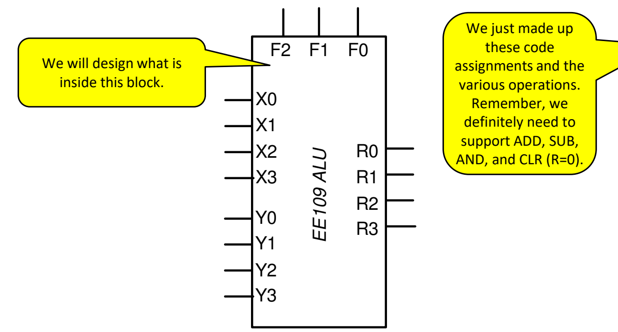
```
• ADD  k   (ACC += k)
• SUB  k   (ACC -= k)
• AND  k   (ACC &= k)
• LOAD k   (ACC = k)
• CLR      (ACC = 0)
• OUT      (OUT = ACC)
```

Defining the Machine Code

- Machine code refers to the ____________ representation of each instruction.
- We first need to define the actual opcodes so we can translate the assembly you wrote on the previous slide into binary for the hardware to execute
- Before we do that, let's consider the hardware design as this will help us choose appropriate opcodes

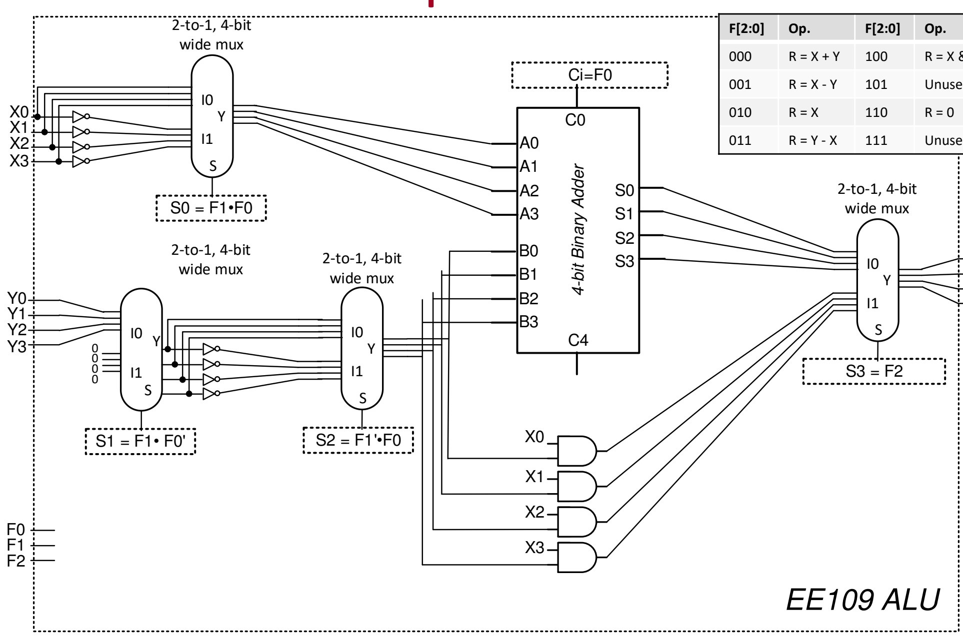
Arithmetic and Logic Unit

- Let's use the ALU we designed in a previous unit…



F[2:0]	Op./Result
000	R = X + Y
001	R = X - Y
010	R = X
011	R = Y - X
100	R = X & Y
101	Unused
110	R = 0
111	Unused

Completed ALU



F[2:0]	Op.	F[2:0]	Op.
000	R = X + Y	100	R = X & Y
001	R = X - Y	101	Unused
010	R = X	110	R = 0
011	R = Y - X	111	Unused

Control Logic

R	FS[2:0]	S0	S1	S2	Ci	S3
X+Y	000	0	0	0	0	0
X-Y	001	0	0	1	1	0
X	010	0	1	0	0	0
Y-X	011	1	0	0	1	0
X & Y	100	0	0	0	d	1
unused	101	d	d	d	d	d
0	110	d	1	0	d	1
unused	111	d	d	d	d	d

- S0 = F1•F0
- S1 = F1•F0'
- S2 = F1'•F0
- Ci = F0
- S3 = F2

S2

F2F1 \ F0	0	1
00		1
01		
11		d
10		d

Ci

F2F1 \ F0	0	1
00		1
01		1
11	d	d
10	d	d

S0

F2F1 \ F0	0	1
00		
01		1
11	d	d
10		d

S1

F2F1 \ F0	0	1
00		
01	1	
11	1	d
10		d

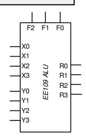
S3

F2F1 \ F0	0	1
00		
01		
11	1	d
10	1	d

Defining the Machine Code Format

- Using the ALU design can you suggest opcodes for the various instructions?
  - The accumulator (ACC) will be connected to the result of the ALU
  - But should the ACC be connected to the X or Y input of the ALU?
    - Important: We achieve Load by passing __ through the ALU to the ACC, so we need the constant to come in on X (so ______ cannot)

Instruction Set Summary
- ADD k (ACC += k)
- SUB k (ACC -= k)
- AND k (ACC &= k)
- LOAD k (ACC = k)
- CLR (ACC = 0)
- OUT (OUT = ACC)



+

F[2:0]	Op./Result
000	R = X + Y
001	R = X - Y
010	R = X
011	R = Y - X
100	R = X & Y
101	Unused
110	R = 0
111	Unused

=

Instruc.	OPCODE	Op./Result
	000	
	001	
	010	
	011	
	100	
	101	
	110	
	111	

Assembler

- Now translate the assembly you found from a few slides back to machine code and show it as 2 hex digits per instruction
- The "high-level" code was
  (7 - 4 + 6) & 3
- "Compile" it to an appropriate instruction sequence (i.e. assembly)
  - CLR = ______
  - ADD 7 = ______
  - SUB 4 = ______
  - ADD 6 = ______
  - AND 3 = ______

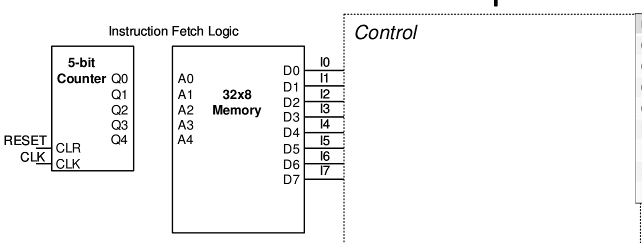
Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C
OUT	001	OUT = ACC
LOAD	010	ACC = C
SUB	011	ACC = ACC - C
AND	100	ACC = ACC & C
-	101	Unused
CLR	110	ACC = 0
-	111	Unused

7 6 5	4	3 2 1 0
Opcode (3-bits)	Unused	Constant (4-bits)

Chosen Instruction Format

Processor Datapath

- Now let's consider the processor data path



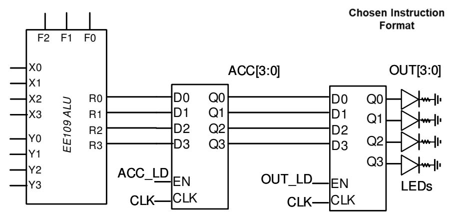
F[2:0]	Op./Result
000	R = X + Y
001	R = X - Y
010	R = X
011	R = Y - X
100	R = X & Y
101	Unused
110	R = 0
111	Unused

Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C
OUT	001	OUT = ACC
LOAD	010	ACC = C
SUB	011	ACC = ACC - C
AND	100	ACC = ACC & C
-	101	Unused
CLR	110	ACC = 0
-	111	Unused

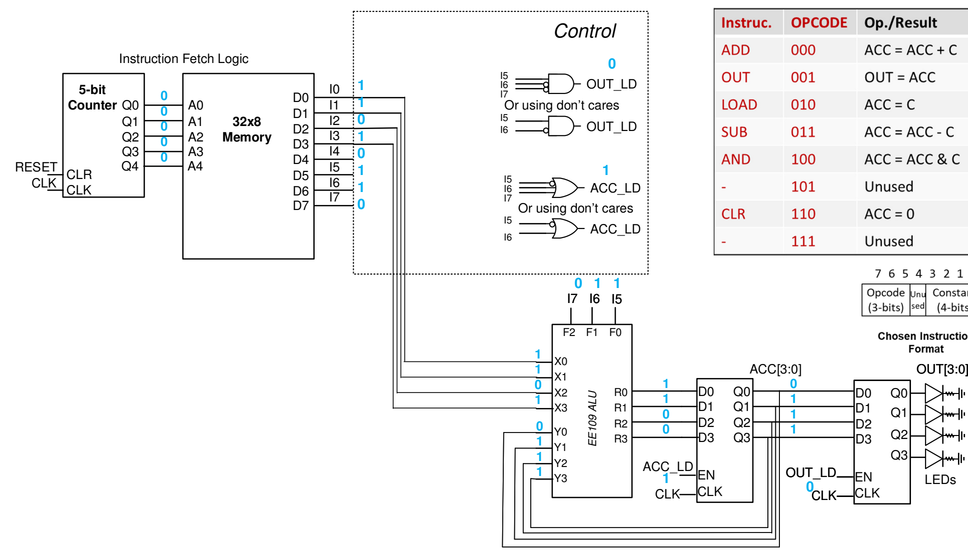
7 6 5	4	3 2 1 0
Opcode (3-bits)	Unused	Constant (4-bits)

Chosen Instruction Format

R	I[7:5]	ACC_LD	OUT_LD
ACC = ACC + C	000		
OUT = ACC	001		
ACC = C	010		
ACC = ACC - C	011		
ACC = ACC & C	100		
Unused	101		
ACC = 0	110		
Unused	111		



Sample Execution of SUB 11



Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C
OUT	001	OUT = ACC
LOAD	010	ACC = C
SUB	011	ACC = ACC - C
AND	100	ACC = ACC & C
-	101	Unused
CLR	110	ACC = 0
-	111	Unused

7 6 5	4	3 2 1 0
Opcode (3-bits)	Unused	Constant (4-bits)

Chosen Instruction Format

A Problem

- Write assembly for:
  - ((7 & 3) + (6 & 5))

- ______________________________

A Solution

- Let's modify our processor as follows:
  - Add _____________ for temporary storage: ______________
    - Could add more but we'll keep it simple
  - A new instruction to save the ______ to a register:
    SAVE Rx (______________)
  - Update ALU instructions to be able to specify a ______________ rather than just a constant
    ADD Rx (ACC = ACC + Rx)
    SUB Rx (ACC = ACC - Rx)
    AND RX (ACC = ACC & Rx)
    LOAD Rx (ACC = Rx)
  - Update the instruction format to use the leftover bit to indicate whether the operand is a constant or should come from a register

7 6 5	4	3 2 1 0
Opcode (3-bits)		Constant (4-bits)
Opcode (3-bits)		Unused (3-bits)

New Instruction Format

Updated Assembly

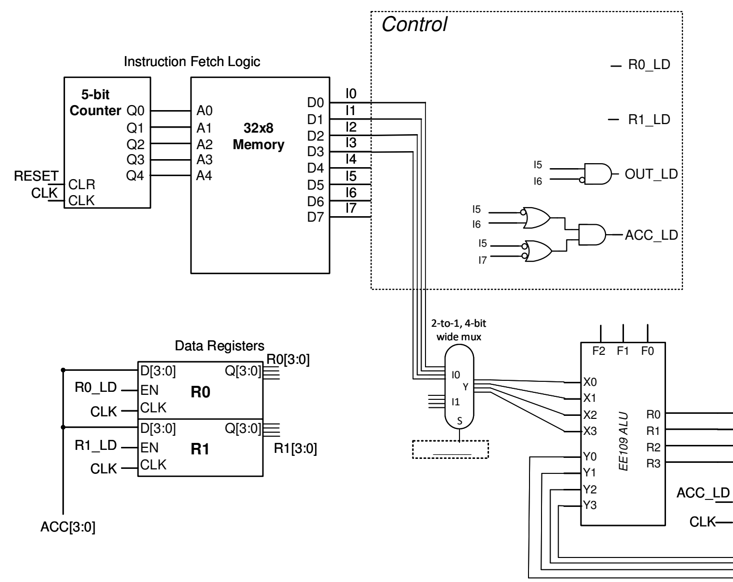
- Write assembly for:
  - ((7 & 3) + (6 & 5))
- New assembly & machine code

Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C/R
OUT	001	OUT = ACC
LOAD	010	ACC = X
SUB	011	ACC = ACC - C/R
AND	100	ACC = ACC & C/R
-	101	Unused
CLR	110	ACC = 0
SAVE Rx	111	Rx = ACC

7 6 5	4	3 2 1 0	
Opcode (3-bits)	C/R 0	Constant (4-bits)	
Opcode (3-bits)	C/R 1	Unused (3-bits)	Reg 0/1

New Instruction Format

Updated Processor Datapath

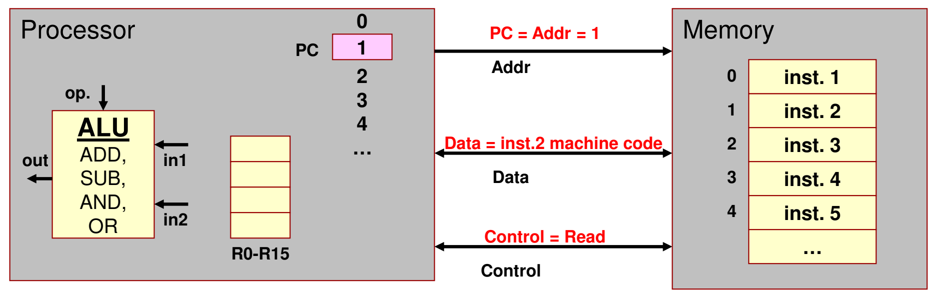


Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C/R
OUT	001	OUT = ACC
LOAD	010	ACC = X
SUB	011	ACC = ACC - C/R
AND	100	ACC = ACC & C/R
-	101	Unused
CLR	110	ACC = 0
SAVE Rx	111	Rx = ACC

ADDING A JUMP/CONTROL

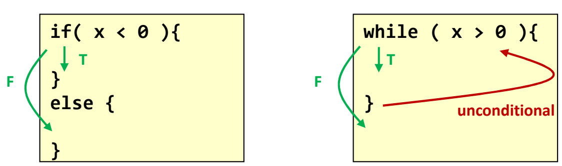
Recall: PC and Sequential Execution

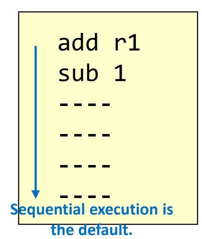
- By default, high level code and assembly instructions execute ____________________.
- The PC (__________________) stores the address of the instruction we will fetch next from memory
  - **Sequential** execution results because the PC simply counts (0,1,2,3,…)
  - But what if we want to skip forward or backward in our code?
  - We need ________ instructions

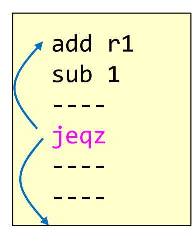


Program Flow

- We need some way to implement conditionals (`if` statements) and loops (`for` / `while`) , which require "jumping" to non-sequential instructions
- Sometimes jumping is _______________ (only if a condition is true) and other time it is ____________ (always)
- In assembly, a **jump** (aka branch) instruction updates the PC (program counter) so that we fetch some new instruction
  - Let us add a `jeqz` instruction (jump if ACC ____________)







Use special instructions to jump to non-sequential locations (backward = loop, forward = conditional)

JEQZ

- We'll assign our one unused opcode for the new jump instruction, `jeqz`
  - We will use the lower 5 bits of the instruction for the new PC value
- Description: Jump to a new PC value IF the accumulator equals 0

```
if(ACC == 0)
  PC = ______ // jump
else
  PC = ______ // sequential
```

- Uses the result from the previous instruction (which is in the ACC) to compare to 0 (e.g. SUB 1, JEQZ *npc*)
- How can we achieve an unconditional jump (so that I jump for sure)

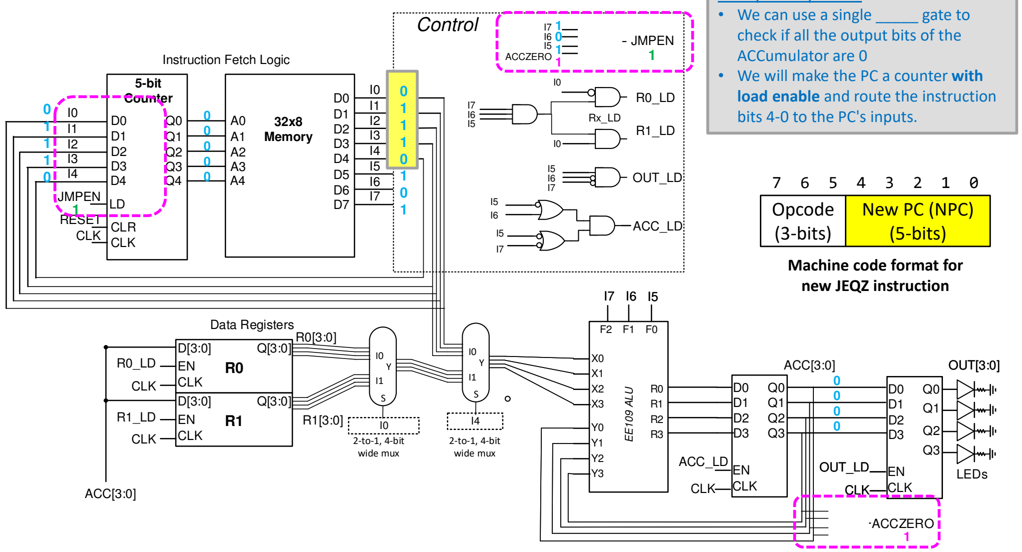
____
JEQZ *npc*

Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C/R
OUT	001	OUT = ACC
LOAD	010	ACC = C/R
SUB	011	ACC = ACC - C/R
AND	100	ACC = ACC & C/R
JEQZ	101	If ACC=0, PC = NPC
CLR	110	ACC = 0
SAVE Rx	111	Rx = ACC

7 6 5	4 3 2 1 0
Opcode (3-bits)	New PC (NPC) (5-bits)

Machine code format for new JEQZ instruction

Data & Control for JEQZ Instruction



Datapath updates

- We can use a single _____ gate to check if all the output bits of the ACCumulator are 0
- We will make the PC a counter **with load enable** and route the instruction bits 4-0 to the PC's inputs.

7 6 5	4 3 2 1 0
Opcode (3-bits)	New PC (NPC) (5-bits)

Machine code format for new JEQZ instruction

Example

- Examine the C code below, implementing a loop to sum the first n=5 integers. Complete the corresponding assembly
- Tips:
  - Allocate variables to specific registers (R0, R1)
  - A statement like x += y requires a 3-instruction sequence:
    LOAD x (to the ACC),
    ADD y (to the ACC),
    SAVE x (from ACC to x)

C Code

```
1 int s=0; // R0
  int i;   // R1
2 for(i=5; i != 0; i--)
           3      5   6
  {
4   s += i;
  }
7 print(s)
```

```
1 00: CLR
  01: SAVE R0
2 02: LOAD 5
  03: SAVE R1
  04: LOAD R1
3 05: JEQZ __
4 06: ________
  07: ________
  08: ________
5 09: ________
  0a: ________
  0b: ________
  0c: CLR
6 0d: JEQZ __
7 0e: LOAD R0
  0f: OUT
```

Corresponding Assembly

OTHER INSTRUCTION SETS…

Historical Instruction Format Options

- Instruction sets limit the _________________ used in an instruction due to…
  - To limit the complexity of the hardware
  - So that when an instruction is coded to binary it ________ in a certain # of bits
- Different instruction sets specify these differently
  - 3 operand instruction set (____________) -> (32-bit processors)
    - Usually all 3 operands in registers
    - Format: ADD DST, SRC1, SRC2 (DST = SRC1 + SRC2)
  - 2 operand instructions (________)
    - Second operand doubles as source and destination
    - Format: ADD SRC1, S2/D (S2/D = SRC1 + S2/D)
  - 1 operand instructions (Low-End Embedded, Java Virtual Machine)
    - Implicit operand to every instruction usually known as the Accumulator (or ACC) register
    - Format: ADD SRC1 (ACC = ACC + SRC1)
  - 0 operand instructions / stack architecture
    - Push operands on a stack: PUSH X, PUSH Y
    - ALU operation: ADD (Implicitly adds top two items on stack: X + Y & replaces them with the sum)

General Instruction Format Issues

- Consider the high-level code
  - F = X + Y – Z
  - G = A + B
- Simple embedded computers often use single operand format
  - Smaller data size (8-bit or 16-bit machines) means limited instruction size
- Modern, high performance processors (Intel, ARM) use 2- and 3-operand formats

Three-Operand	Two-Operand	Single-Operand	Stack Arch.

ADD F,X,Y
SUB F,F,Z
ADD G,A,B
		LOAD A
		ADD B
		STORE G
			PUSH Z
			PUSH Y
			SUB
			PUSH X
			ADD
			POP F

(+) More natural program style (+) Smaller instruction count			(+) Smaller size to encode each instruction

MORE PRACTICE

More Practice (On Own Time)

- Write assembly for:
  - `( (4&14) + (5&3) - (6&11) + (8&13))`
- Try to use as few instructions as you can

Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C/R
OUT	001	OUT = ACC
LOAD	010	ACC = X
SUB	011	ACC = ACC - C/R
AND	100	ACC = ACC & C/R
-	101	Unused
CLR	110	ACC = 0
SAVE Rx	111	Rx = ACC

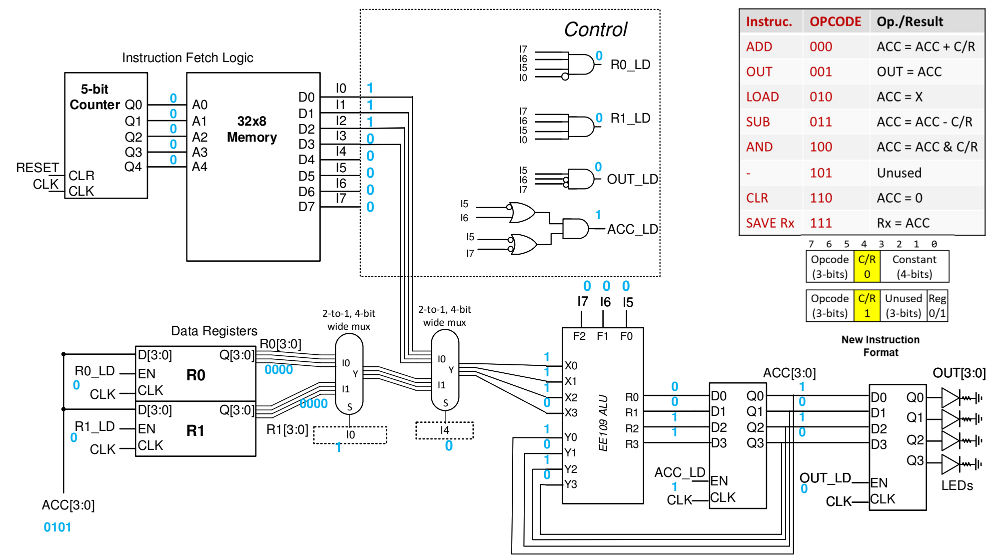
7 6 5 4 3 2 1 0

Opcode (3-bits)	C/R 0	Constant (4-bits)

Opcode (3-bits)	C/R 1	Unused (3-bits)	Reg 0/1

New Instruction Format

ADD 7



Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C/R
OUT	001	OUT = ACC
LOAD	010	ACC = X
SUB	011	ACC = ACC - C/R
AND	100	ACC = ACC & C/R
-	101	Unused
CLR	110	ACC = 0
SAVE Rx	111	Rx = ACC

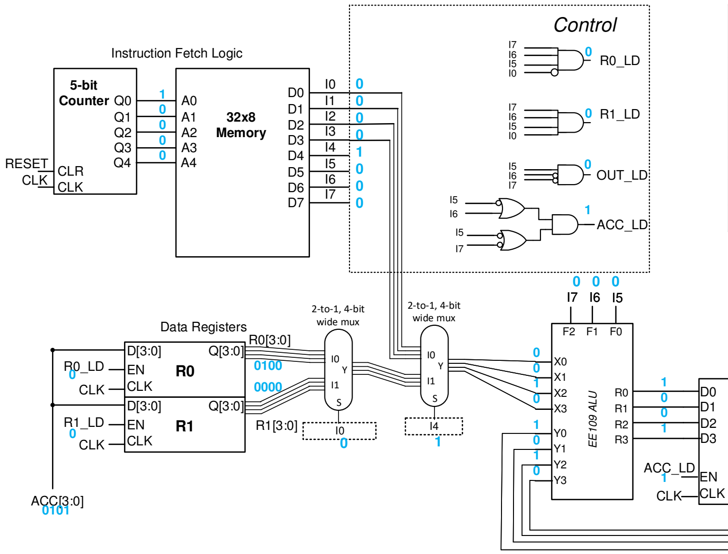
7 6 5 4 3 2 1 0

Opcode (3-bits)	C/R 0	Constant (4-bits)

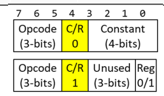
Opcode (3-bits)	C/R 1	Unused (3-bits)	Reg 0/1

New Instruction Format

ADD R0

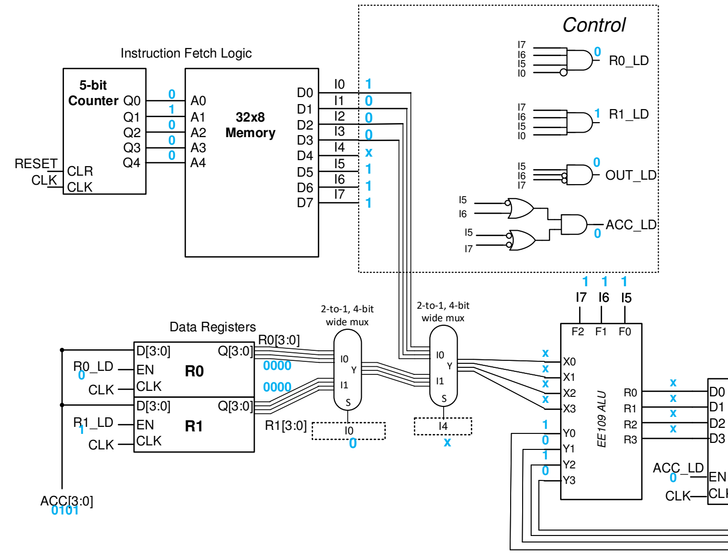


Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C/R
OUT	001	OUT = ACC
LOAD	010	ACC = X
SUB	011	ACC = ACC - C/R
AND	100	ACC = ACC & C/R
-	101	Unused
CLR	110	ACC = 0
SAVE Rx	111	Rx = ACC

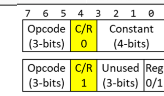


New Instruction Format

SAVE R1



Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C/R
OUT	001	OUT = ACC
LOAD	010	ACC = X
SUB	011	ACC = ACC - C/R
AND	100	ACC = ACC & C/R
-	101	Unused
CLR	110	ACC = 0
SAVE Rx	111	Rx = ACC



New Instruction Format