

Unit 16

Computer Organization
Design of a Simple Processor

INSTRUCTION SET OVERVIEW

Instruction Set Overview

- The instruction set defines the software interface to the processor and memory system
- Most processors define their own *unique* instruction set unless they are trying to be compatible with some other vendor
 - Which means code compiled for one processor will not run on another
- Instruction set is the vocabulary 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 machine code) that control the processor

Components of the Instruction Set

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

Instruction Set Architecture (ISA)

- 2 instruction set approaches
 - CISC = Complex instruction set computer
 - Large, rich set of instructions (vocabulary)
 - Useful when programmers actually wrote assembly code
 - More work per instruction, slower clock cycle
 - RISC = Reduced instruction set computer
 - Small, basic, but *sufficient* instruction set (vocabulary)
 - With maturation of compilers, 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

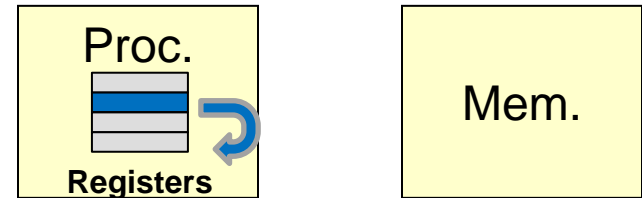
Kinds of Instructions

- Most assembly/machine instructions fall into one of **three** categories
- **Arithmetic/Logic**
 - Example: `ADD r1, r2, r3 // r1 = r2 + r3`
- **Data Transfer (to and from memory)**
 - Example: `LOAD r1, addr // set register 1 to the value in mem. at addr`
- **Control**
 - 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**"

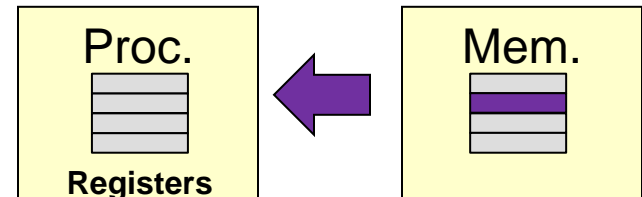
Operands

- Addressing modes refers to how an instruction specifies where the operands are
- Can be in a
 - **register**,
 - **memory location**, or a
 - **constant** that is part of the instruction itself (**aka. immediate 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

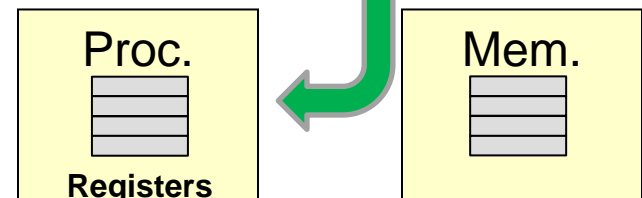
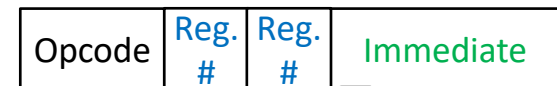
ADD r1, r2, r3



LOAD r5, 0x1ac8



ADD r5, r3, immediate



DESIGN OF A SIMPLE INSTRUCTIONS SET AND PROCESSOR

What Shall We Do?

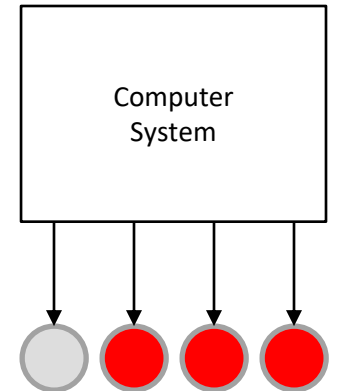
- Let's design a simple processor to understand the entire flow from writing software to designing the hardware
 - This may not be the most advanced processor but the goal is to give you a fully working example from software to hardware

The Instruction Set (1)

- To start we will define the instruction set
- Let's use 4-bit data values (i.e. all data operands will be 4-bits)
- Let's make this a simple calculator-like processor that can perform at least the following 3 operations:
 - ADD
 - SUB
 - AND
- 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 ACCumulator-based architecture where the ACC register is always one implied operand
 - ADD 7 means: $ACC += 7$
 - SUB 5 means: $ACC -= 5$

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:
 - Load constant (Acc = const)
 - Clear (Acc = 0)
 - Out (OUT = Acc)
- That leaves us with 6 total instructions
 - How many bits do we need for the opcode of our instructions? 3-bits
- 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? 4-bits
- Instructions need 3 opcode + 4 data bits = 7-bits
 - Let's round up to 8-bits for each instruction



Output LEDs
(Display = 7 = 0111₂)

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

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
 - CLR
 - ADD 7
 - SUB 4
 - ADD 6
 - AND 3
- Now we need to convert to binary...

Instruction Set Summary

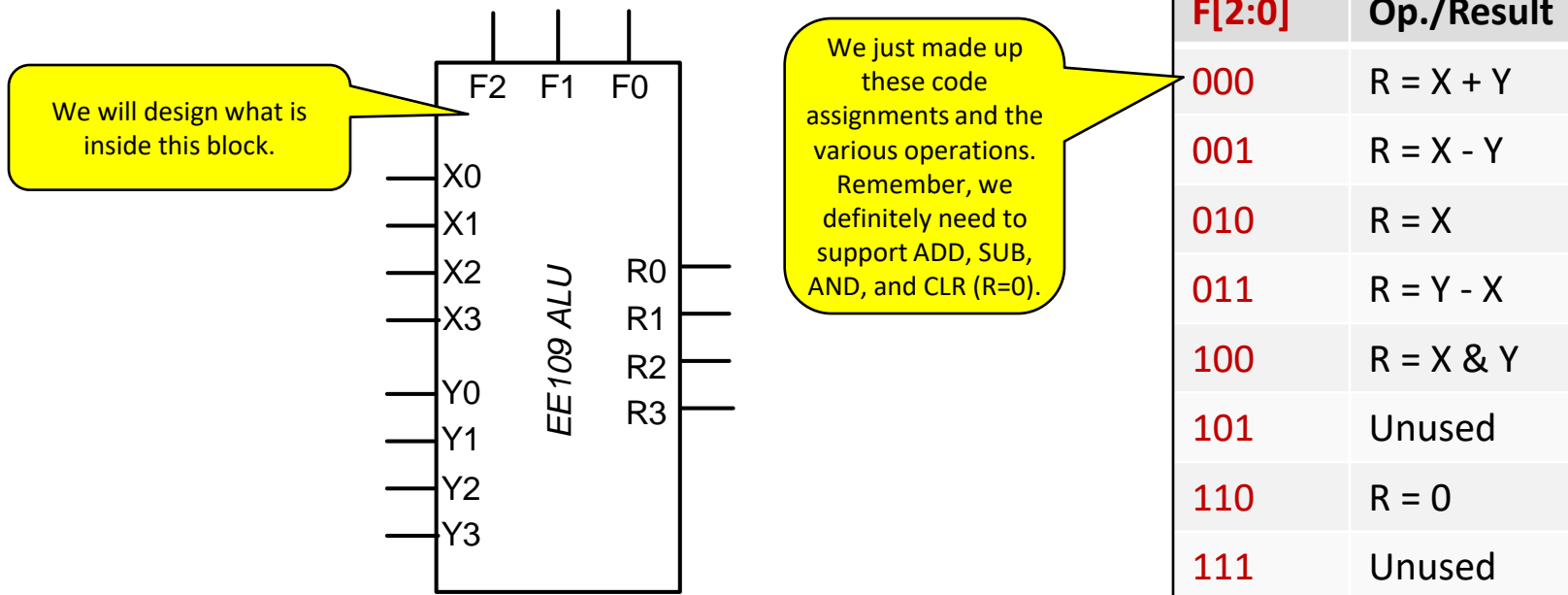
- ADD k (ACC += k)
- SUB k (ACC -= k)
- AND k (ACC &= k)
- LOAD k (ACC = k)
- CLR (ACC = 0)
- OUT (OUT = ACC)

Defining the Machine Code

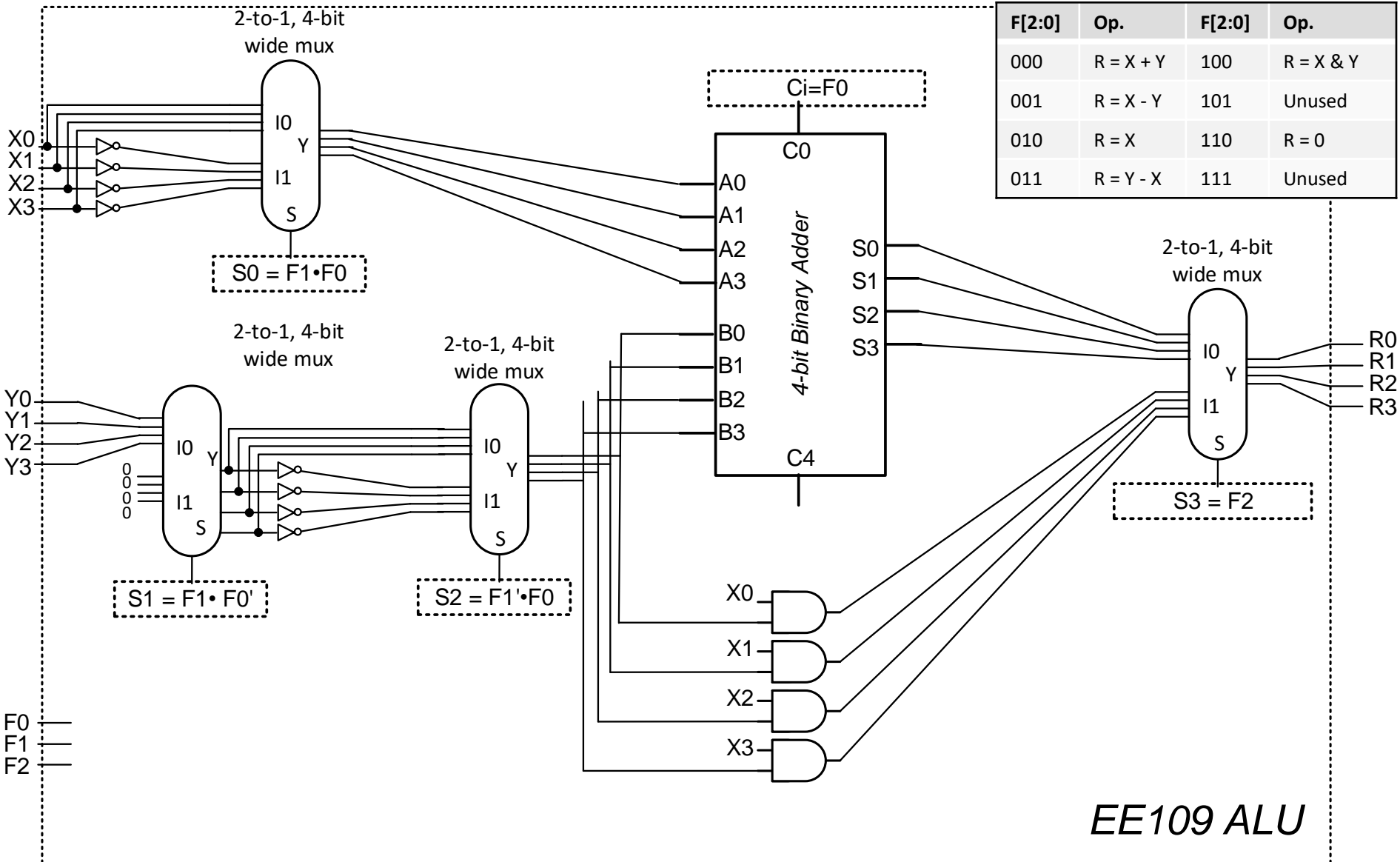
- **Machine code** refers to the binary representation of each instruction.
- We first need to define the actual opcodes so we can translate the assembly you wrote on the previous slide into binary for the hardware to execute
- Before we do that, let's consider the hardware design as this will help us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

Arithmetic and Logic Unit

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

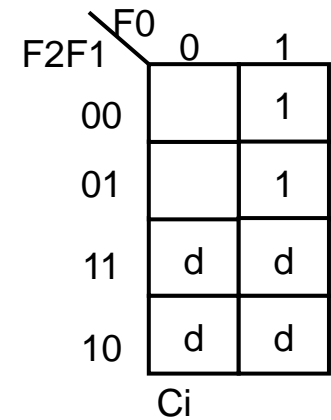
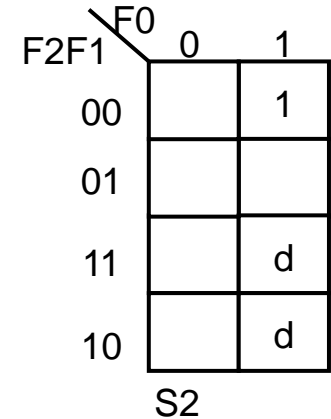


Completed ALU

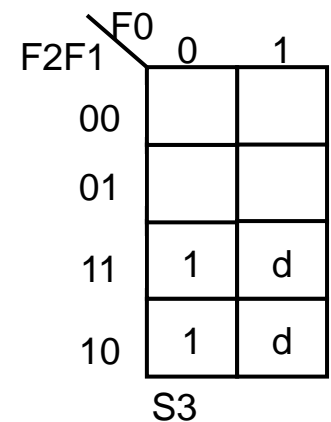
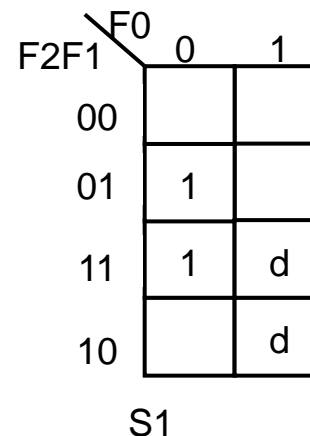
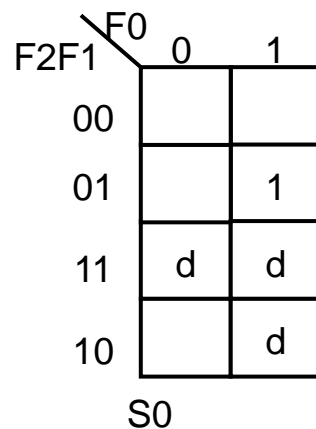


Control Logic

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



- $S0 = F1 \bullet F0$
- $S1 = F1 \bullet F0'$
- $S2 = F1' \bullet F0$
- $Ci = F0$
- $S3 = F2$

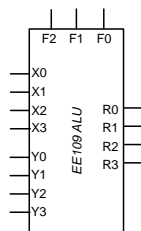


Defining the Machine Code Format

- 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 X through the ALU to the ACC, so we need the constant to come in on X (so ACC 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

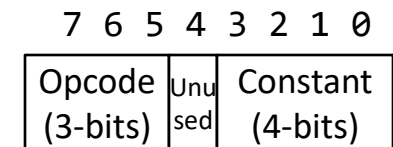
=

Instruc.	OPCODE	Op./Result
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

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 = 0xc0
 - ADD 7 = 0x07
 - SUB 4 = 0x64
 - ADD 6 = 0x06
 - AND 3 = 0x83

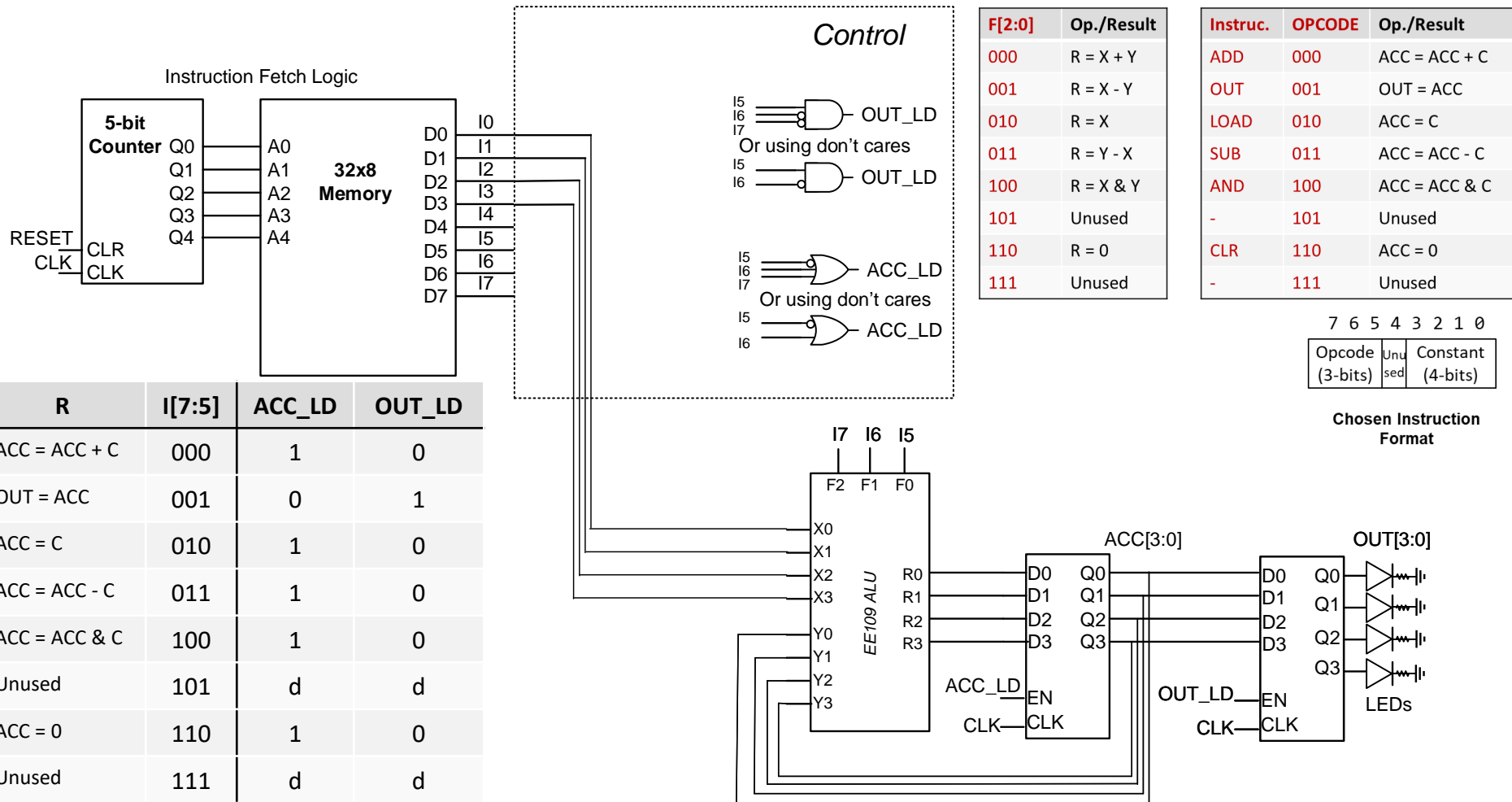
Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C
OUT	001	OUT = ACC
LOAD	010	ACC = C
SUB	011	ACC = ACC - C
AND	100	ACC = ACC & C
-	101	Unused
CLR	110	ACC = 0
-	111	Unused



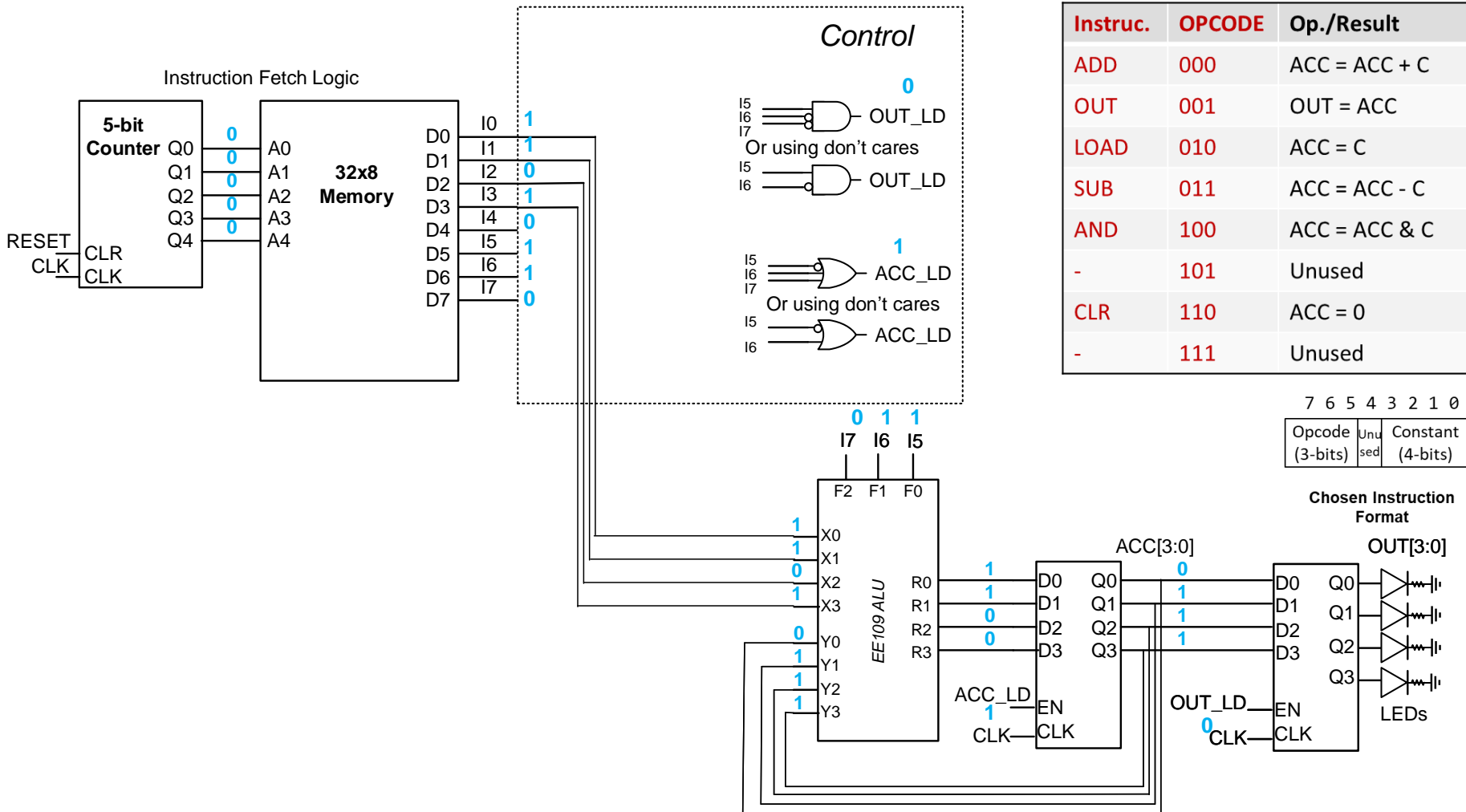
**Chosen Instruction
Format**

Processor Datapath

- Now let's consider the processor data path



Sample Execution of SUB 11



A Problem

- Write assembly for:
 - $((7 \ \& \ 3) + (6 \ \& \ 5))$
 - LOAD 7
 - AND 3
 - No place to store result so we can compute (6&5) separately
- No place to store "temporary" results

A Solution

- Let's modify our processor as follows:
 - Add two registers for temporary storage: R0 and R1
 - Could add more but we'll keep it simple
 - A new instruction to save the ACC to a register:
 SAVE Rx (Rx = ACC)
 - Update ALU instructions to be able to specify a register operand 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

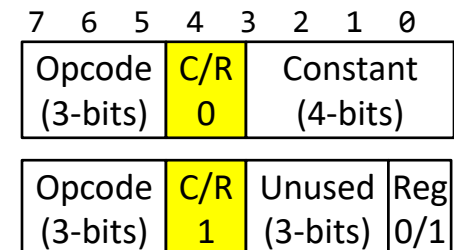
	7	6	5	4	3	2	1	0
Opcode (3-bits)	C/R 0			Constant (4-bits)				
Opcode (3-bits)	C/R 1			Unused (3-bits)			Reg 0/1	

**New Instruction
Format**

Updated Assembly

- Write assembly for:
 - ((7 & 3) + (6 & 5))
- New assembly & machine code
 - LOAD 7 = 0x47
 - AND 3 = 0x83
 - SAVE R1 = 0xf1
 - LOAD 6 = 0x46
 - AND 5 = 0x85
 - ADD R1 = 0x11
 - OUT = 0x20

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
-	101	Unused
CLR	110	ACC = 0
SAVE Rx	111	Rx = ACC



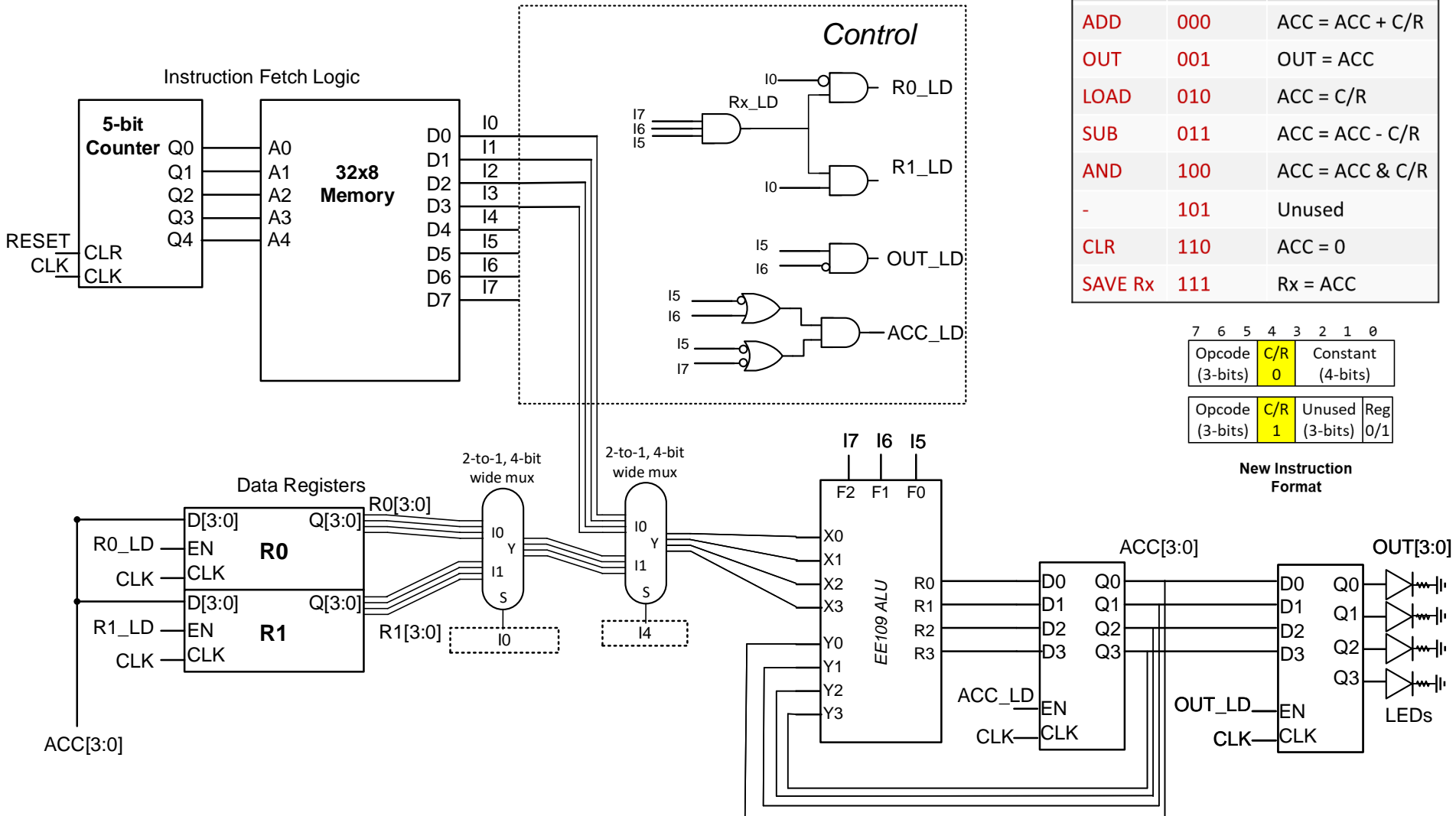
New Instruction
Format

Updated Processor Datapath

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
-	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		

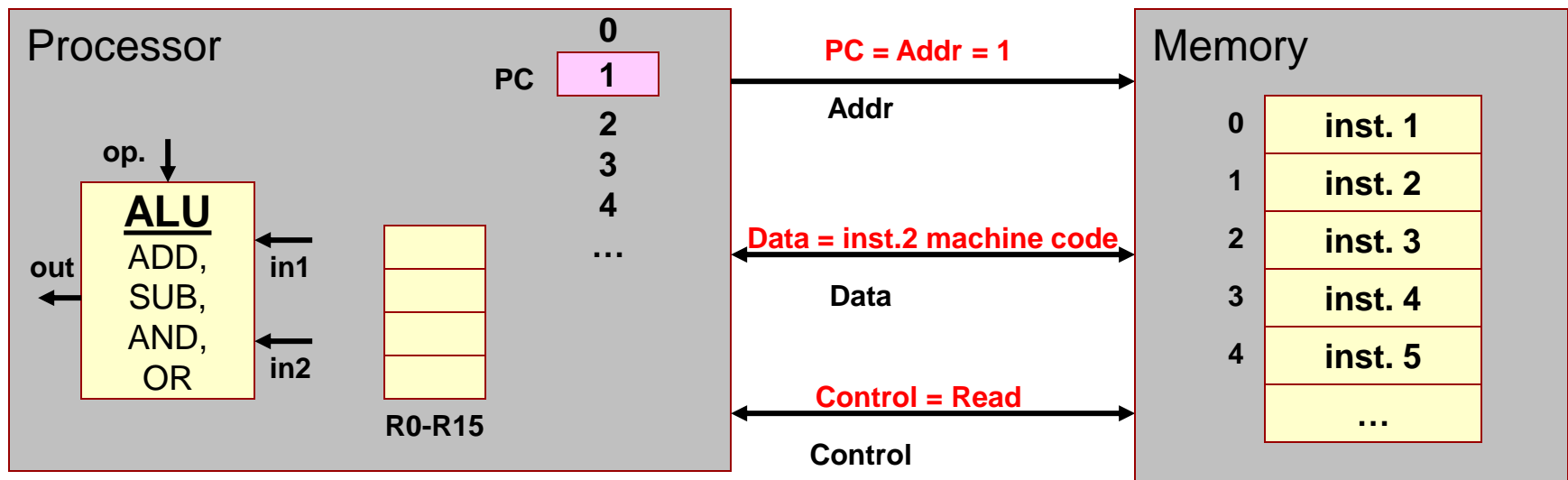
New Instruction Format



ADDING A JUMP/CONTROL

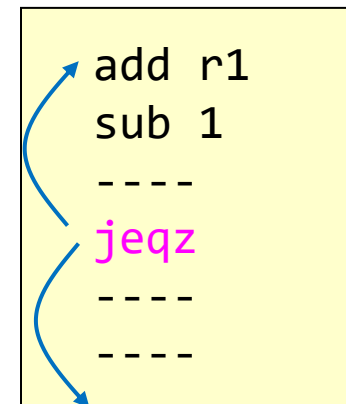
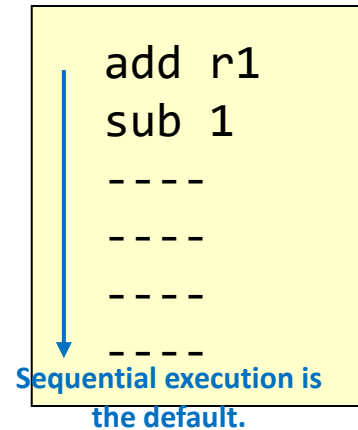
Recall: PC and Sequential Execution

- By default, high level code and assembly instructions execute **sequentially**.
- The PC (Program Counter) 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 **jump** instructions

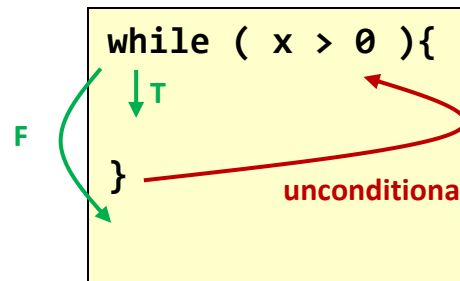
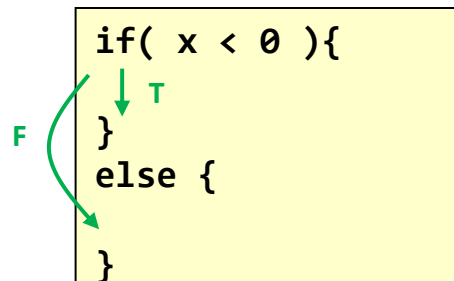


Program Flow

- We need some way to implement conditionals (if statements) and loops (for / while), which require "jumping" to non-sequential instructions
- Sometimes jumping is **conditional** (only if a condition is true) and other time it is **unconditional** (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 equal to zero)



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



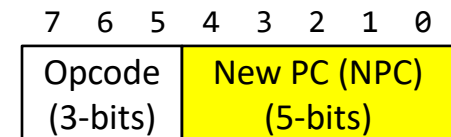
JEQZ

- 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 = NPC // jump
else
    PC = PC + 1 // 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)


```
CLR
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

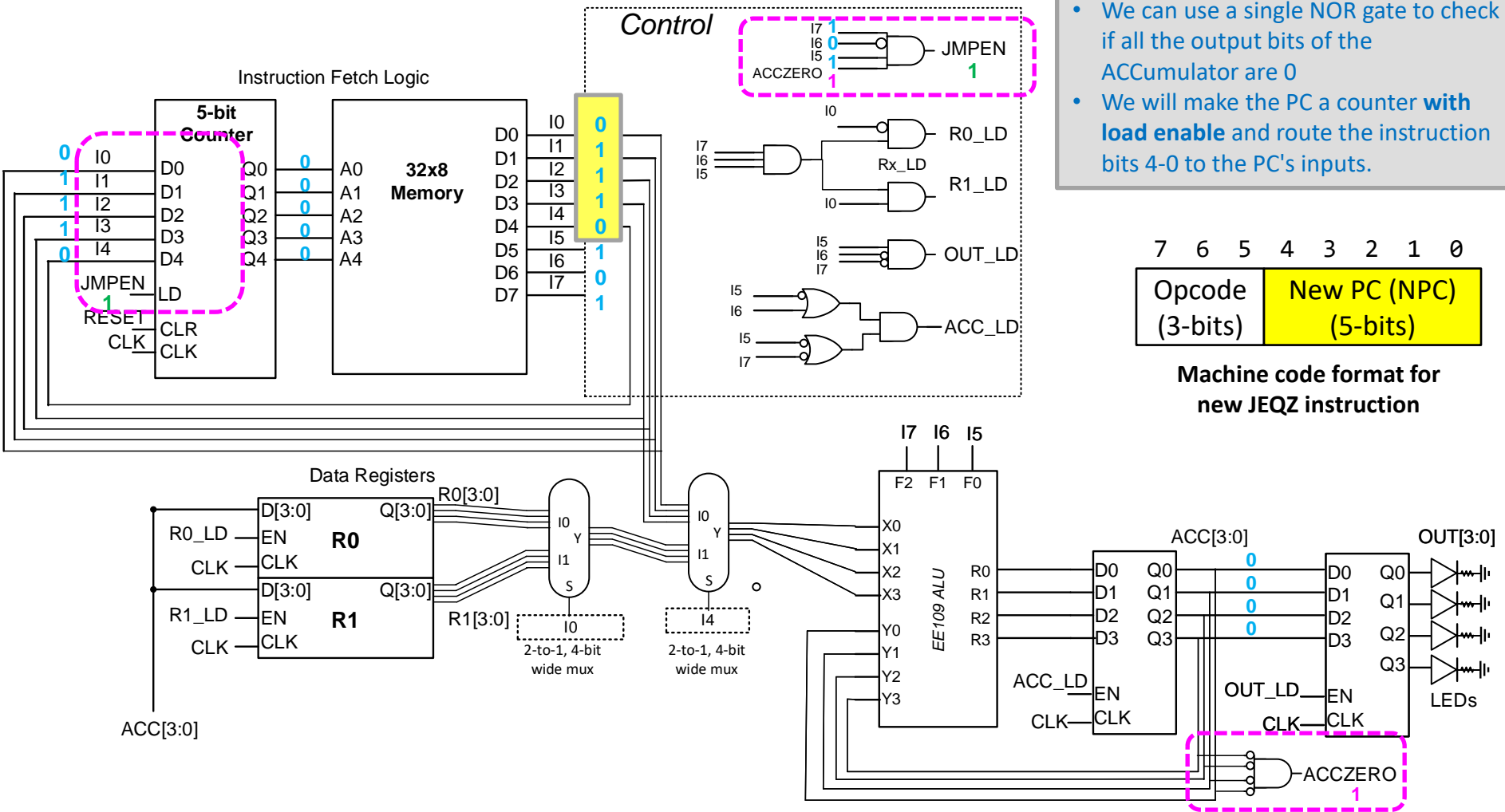


Machine code format for new JEQZ instruction

Data & Control for JEQZ Instruction

Datapath updates

- We can use a single NOR 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.



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

Corresponding Assembly

```

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

OTHER INSTRUCTION SETS...

Historical Instruction Format Options

- Instruction sets limit the number of operands used in an instruction due to...
 - To limit the complexity of the hardware
 - So that when an instruction is coded to binary it can fit in a certain # of bits
- Different instruction sets specify these differently
 - 3 operand instruction set (ARM, PPC) -> (32-bit processors)
 - Usually all 3 operands in registers
 - Format: ADD DST, SRC1, SRC2 (DST = SRC1 + SRC2)
 - 2 operand instructions (Intel / Motorola 68K)
 - 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	MOVE F,X ADD F,Y SUB F,Z MOVE G,A ADD G,B	LOAD X ADD Y SUB Z STORE F 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

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

- LOAD 6
- AND 11
- SAVE R0
- LOAD 4
- AND 14
- SAVE R1
- LOAD 5
- AND 3
- ADD R1
- SAVE R1
- LOAD 8
- AND 13
- ADD R1
- SUB R0
- OUT

Since we can only do ACC – C/R, it means we should already have the sum of the other terms in ACC and then subtract. To compute 6&11 later would then require us to swap in the sum of the other terms into the ACC and then subtract, costing an extra instruction.

Since we have many terms we can use R1 to keep "accumulating" the sum of more terms while we use the ACC to compute the current term.

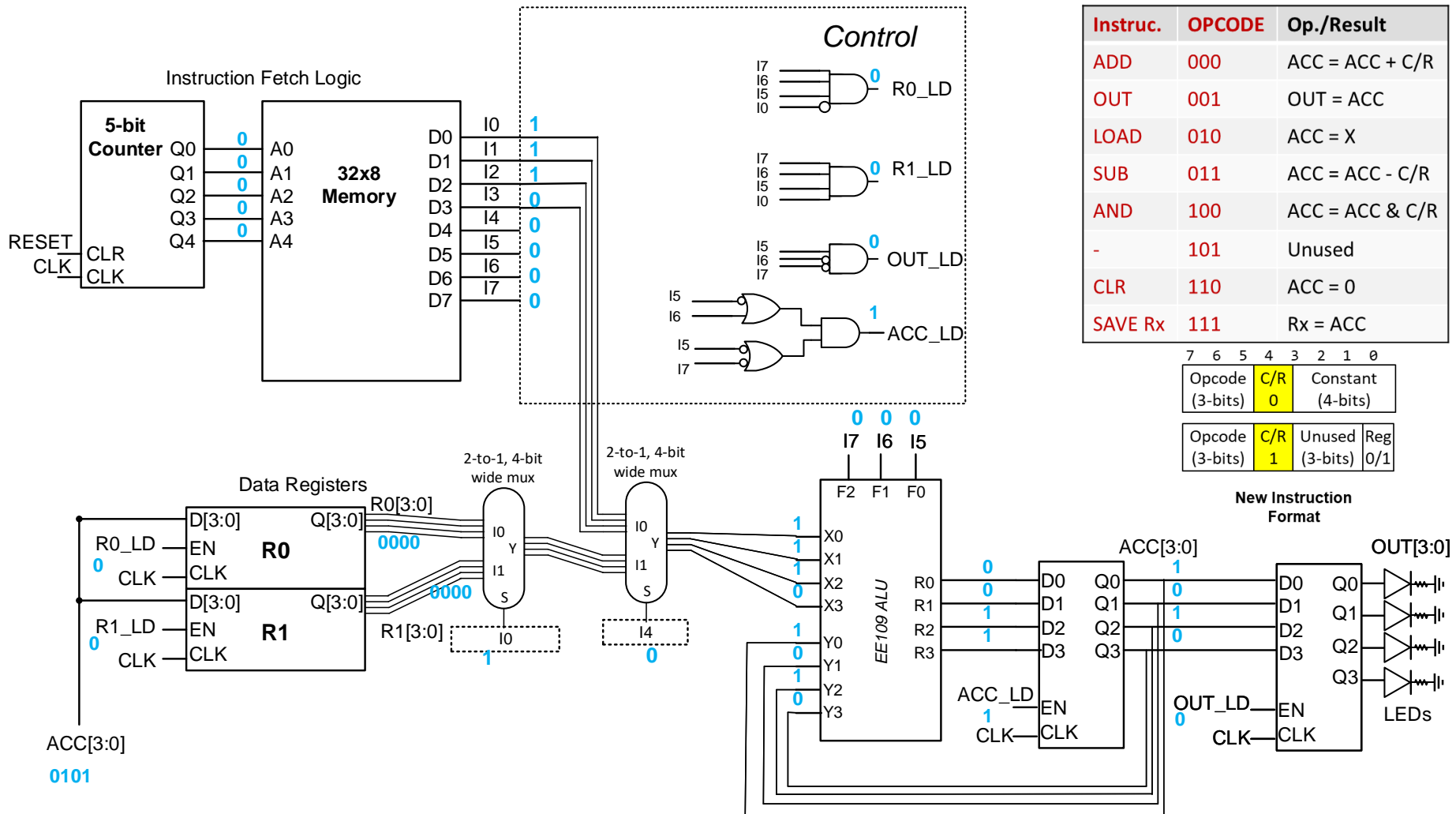
Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C/R
OUT	001	OUT = ACC
LOAD	010	ACC = X
SUB	011	ACC = ACC - C/R
AND	100	ACC = ACC & C/R
-	101	Unused
CLR	110	ACC = 0
SAVE Rx	111	Rx = ACC

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

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

**New Instruction
Format**

ADD 7

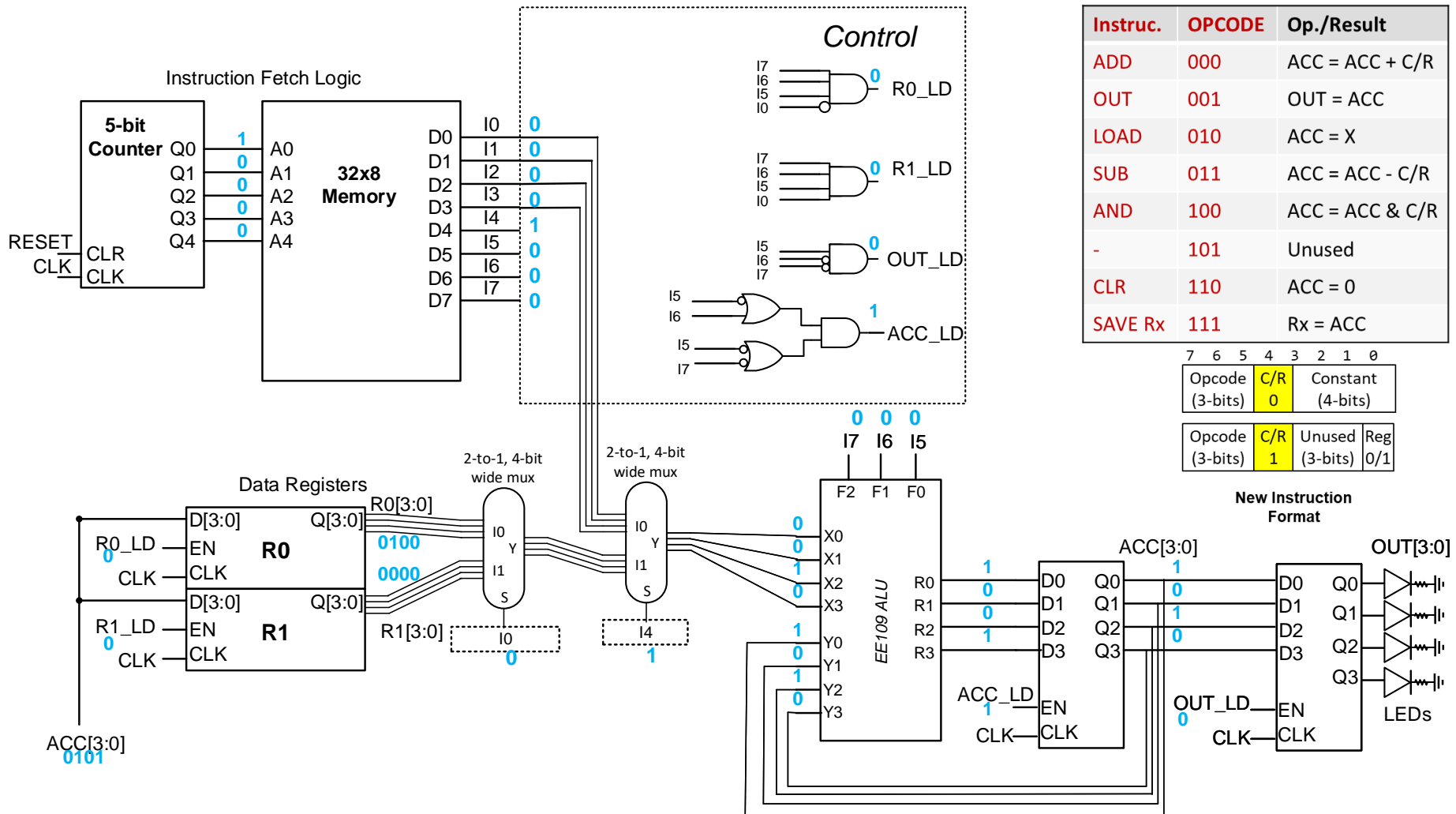


Instruc.	OPCODE	Op./Result
ADD	000	ACC = ACC + C/R
OUT	001	OUT = ACC
LOAD	010	ACC = X
SUB	011	ACC = ACC - C/R
AND	100	ACC = ACC & C/R
-	101	Unused
CLR	110	ACC = 0
SAVE Rx	111	Rx = ACC

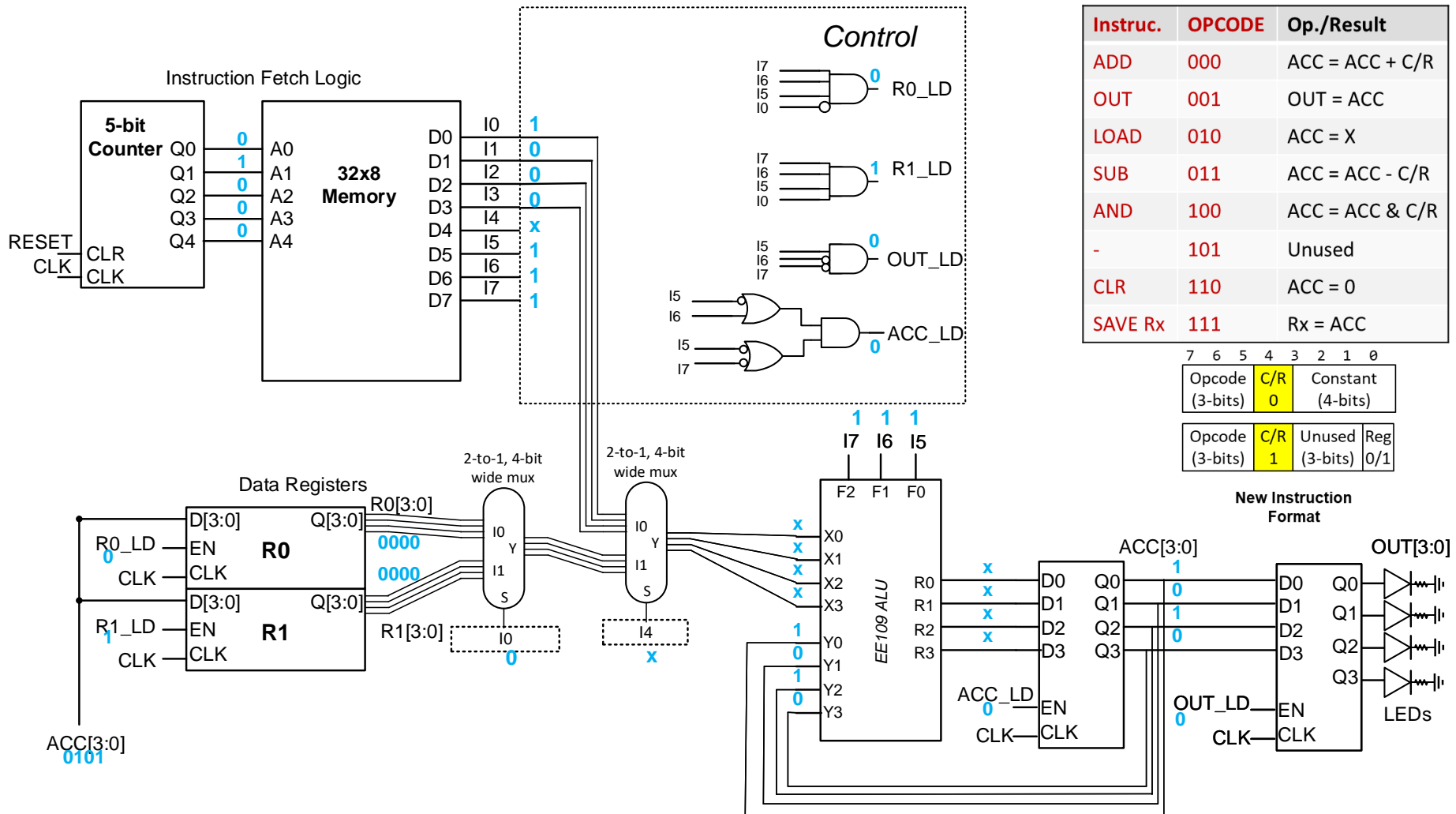
7	6	5	4	3	2	1	0
Opcode (3-bits)			C/R (1-bit)	Constant (4-bits)			
Opcode (3-bits)			C/R (1-bit)	Unused (3-bits)	Reg (0/1)		

New Instruction Format

ADD R0



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

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

New Instruction Format