ECE 428 Programmable ASIC Design

FPGA Microprocessor

Part 1

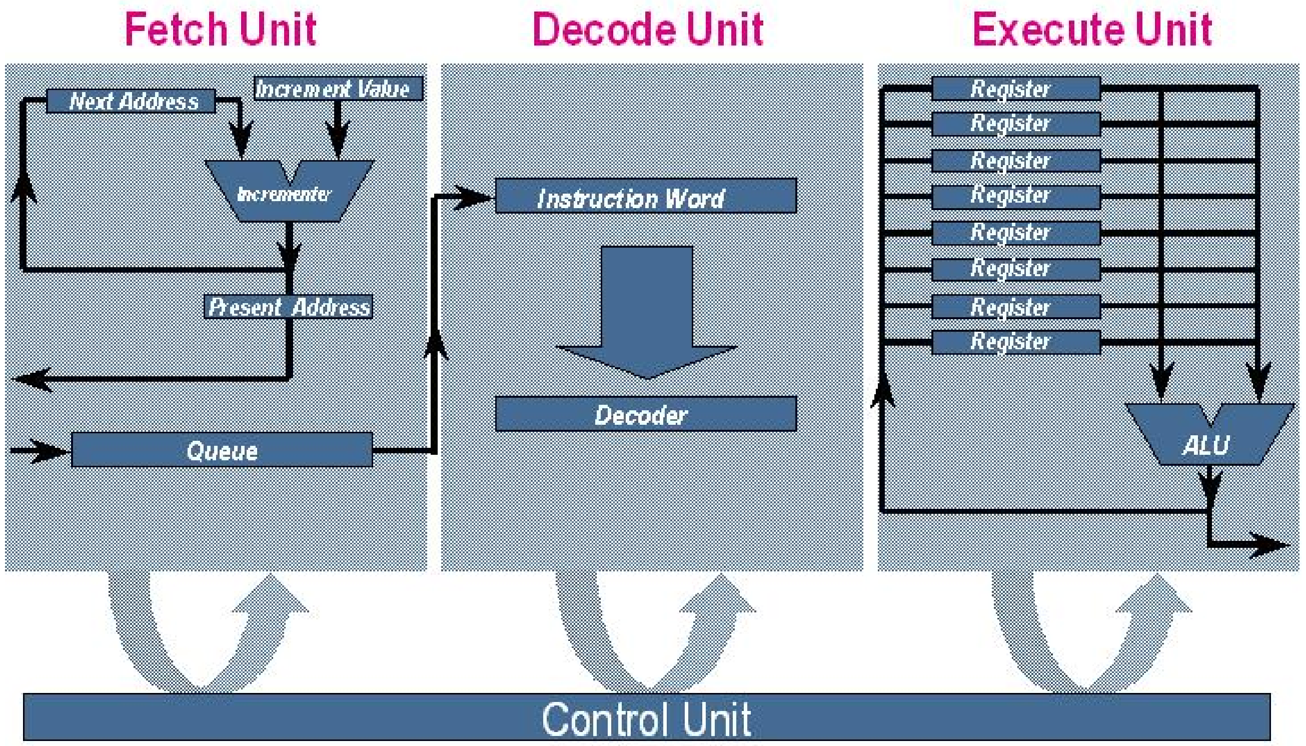
Haibo Wang
ECE Department
Southern Illinois University
Carbondale, IL 62901

Microprocessor ***v.s.*** ASIC

- ❑ For a given function, we can implement it by using a general purpose microprocessor or by designing an ASIC

Microprocessor	ASIC
1. General purpose	1. Application specific
2. Flexible and easy to update by loading new software	2. Less flexible and almost impossible to Update unless the ASIC contains programmable circuits
3. Typically, the performance is not optimized for a given task	3. Normally, optimal performance

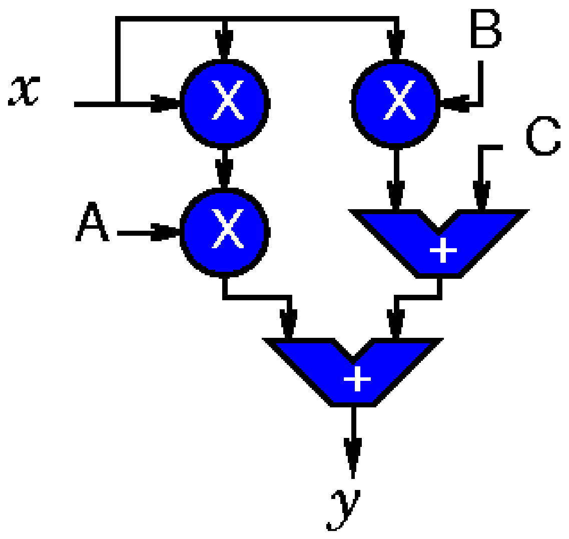
How Does A Microprocessor Work?



- ❑ First, an instruction is fetched into the microprocessor
- ❑ Second, the instruction is decoded for execution
- ❑ Finally, the instruction is executed.

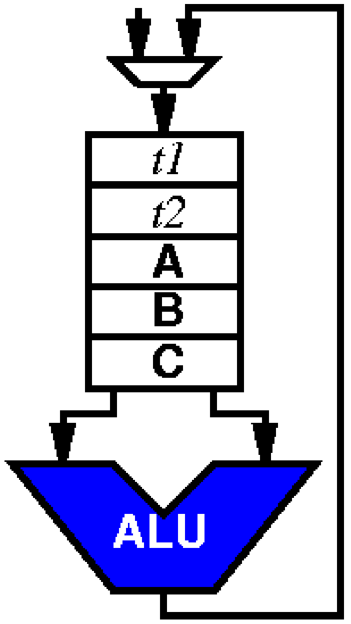
Advantages of FPGA Microprocessors

- The architecture of FPGA microprocessor can be easily modified to achieve optimal performance for a given application.
- Example: calculate Y = A•X^2 + B•X + C



FPGA

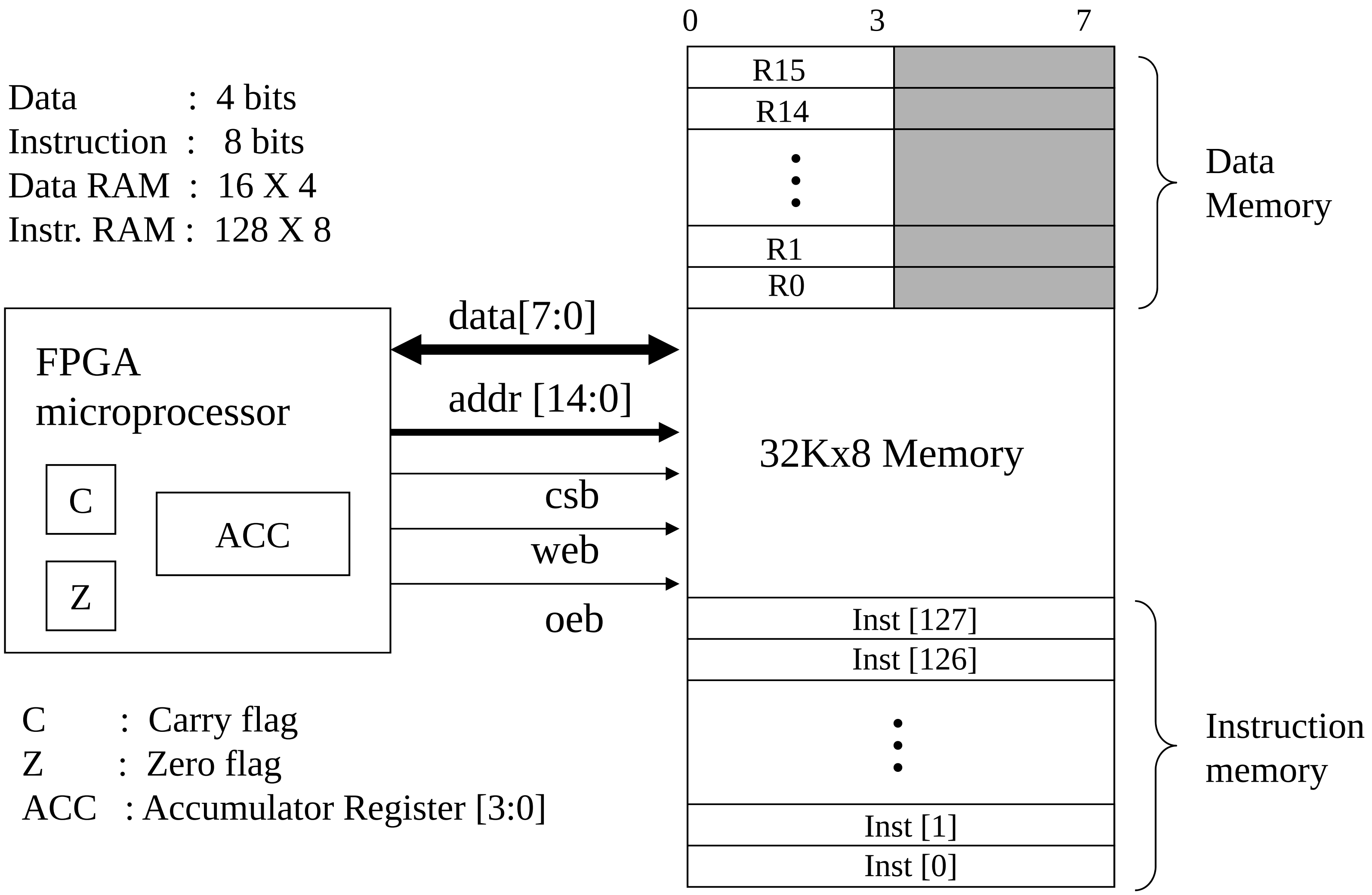
$t1 \leftarrow x$
$t2 \leftarrow \mathbf{A} \times t1$
$t2 \leftarrow t2 + \mathbf{B}$
$t2 \leftarrow \mathbf{t2} \times t1$
$y \leftarrow \mathbf{t2} + \mathbf{C}$



General purpose microprocessor

Introduction to GNOME Microprocessor

Data	: 4 bits
Instruction	: 8 bits
Data RAM	: 16 X 4
Instr. RAM	: 128 X 8



C	: Carry flag
Z	: Zero flag
ACC	: Accumulator Register [3:0]

Instructions of GNOME Microprocessor

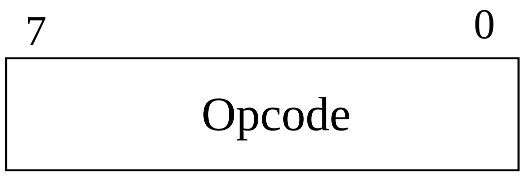
Mnemonic	**Operation**	**Description**
load Rd	ACC $\leftarrow$ Rd	Load the accumulator with the contents of the memory location whose address is d.
loadi d	ACC $\leftarrow$ d	Load the accumulator with the immediate data d
store Rd	Rd ACC	Store the value in the accumulator into RAM location Rd.
add Rd	ACC $\leftarrow$ ACC+Rd+C	Add the content of RAM location Rd and the C flag to the Accumulator.
addi d	ACC $\leftarrow$ ACC+d+C	Add the value d and the C flag to the accumulator.
xor Rd	ACC $\leftarrow$ ACC $\otimes$ Rd	EXCLUSIVE-OR the content of RAM location Rd with the Accumulator.
test Rd	Z $\leftarrow$ ACC $\bullet$ Rd	AND the contents of RAM location Rd with the accumulator, but do not store the results in the accumulator. Instead, set the Z flag if the result is 0, otherwise clear the Z flag.

Instructions of GNOME Microprocessor

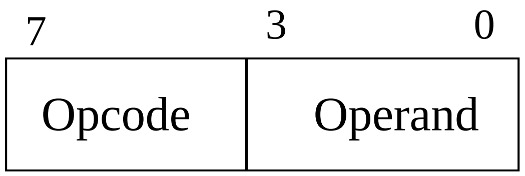
Mnemonic	Operation	Description
clear_c	C ← 0	Clear the C flag to zero.
set_c	C ← 1	Set the C flag to one.
skipc	PC ← PC+1+C	If C=1, skip the next instruction by incrementing the program counter by two instead of one. Otherwise, execute the next instruction
skipz	PC ← PC+1+Z	If Z=1, skip the next instruction by incrementing the program counter by two instead of one. Otherwise, execute the next instruction
jump a	PC ← a	Jump to program address a and execute instructions from address a. a is an address in the range [0..127]

Instruction Encoding

- Instruction Format
  - — Fixed length instructions (8 bits)
  - — Each instruction can have one or zero operand

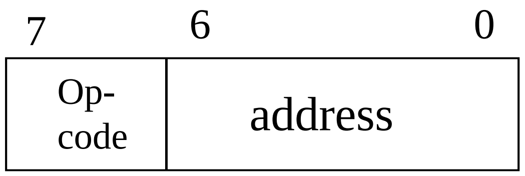


clear_c set_c



load Rd, loadi d, store Rd, add Rd, addi d,

xor Rd, xori d, test Rd, testi d

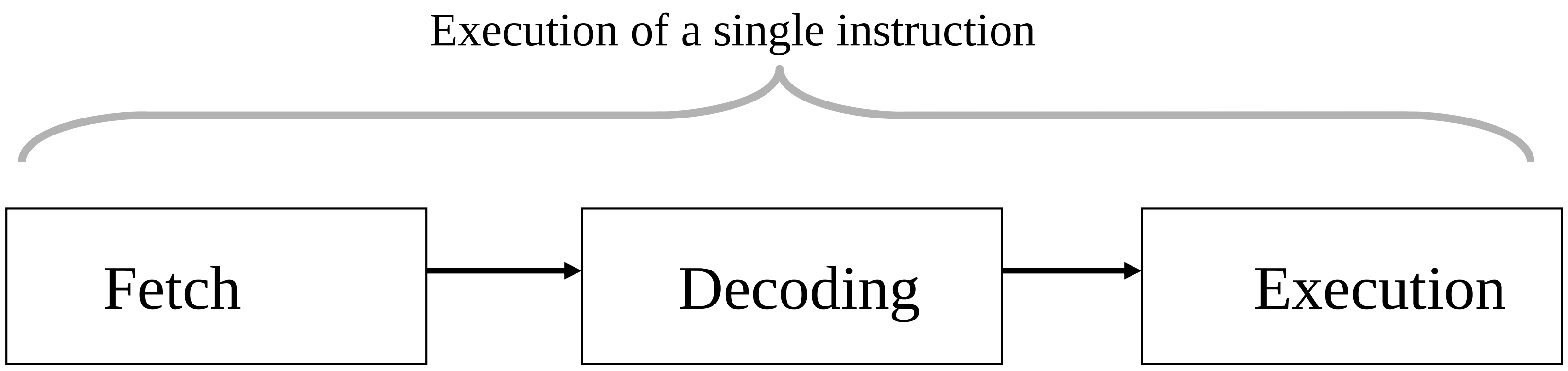


jump a

Instruction Encoding

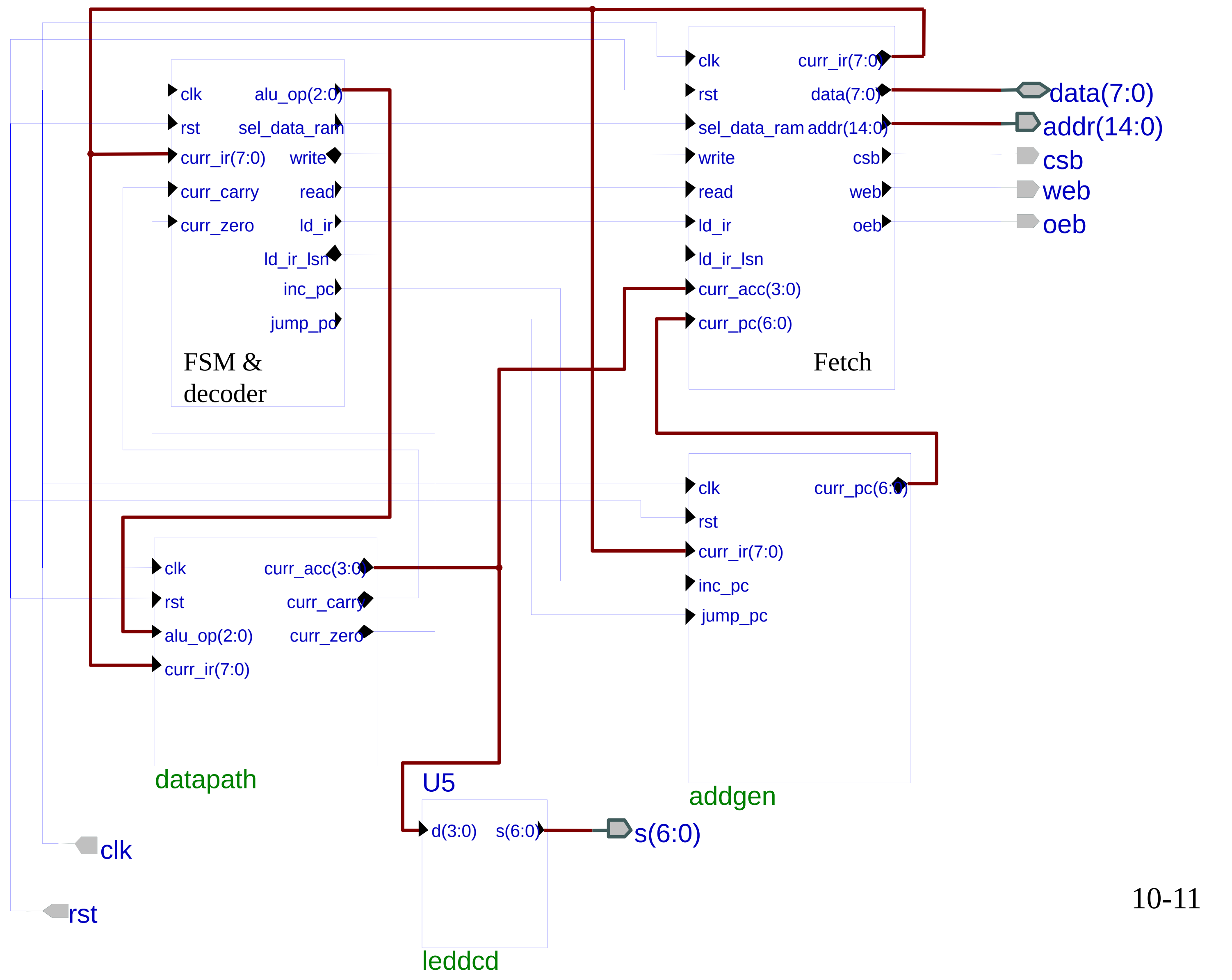
Mnemonic	Encoding	Comment
load Rd	0 1 0 0 d_3 d_2 d_1 d_0	d_3 d_2 d_1 d_0 is the address of RAM location
loadi d	0 0 0 1 d_3 d_2 d_1 d_0	d_3 d_2 d_1 d_0 is the immediate data
store Rd	0 0 1 1 d_3 d_2 d_1 d_0	d_3 d_2 d_1 d_0 is the address of RAM location
add Rd	0 1 0 1 d_3 d_2 d_1 d_0	d_3 d_2 d_1 d_0 is the address of RAM location
addi d	0 0 1 0 d_3 d_2 d_1 d_0	d_3 d_2 d_1 d_0 is the immediate data
xor Rd	0 1 1 0 d_3 d_2 d_1 d_0	d_3 d_2 d_1 d_0 is the address of RAM location
test Rd	0 1 1 1 d_3 d_2 d_1 d_0	d_3 d_2 d_1 d_0 is the address of RAM location
clear_c	0 0 0 0 0 0 0 0	
set_c	0 0 0 0 0 0 0 1	
skipc	0 0 0 0 0 0 1 0	
skipz	0 0 0 0 0 0 1 1	
jump a	1 a_6 a_5 a_4 a_3 a_2 a_1 a_0	a_6 a_5 a_4 a_3 a_2 a_1 a_0 is the new instruction address

Operation of GNOME Microprocessor



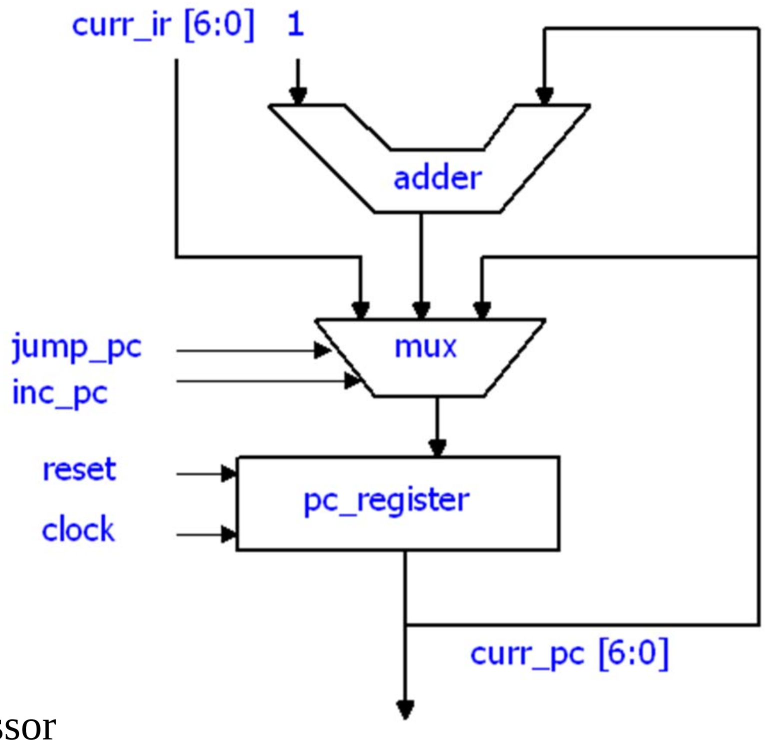
1. Fetch instruction	1. Decode instruction	1. Execute the instruction
2. Update PC register	2. Fetch operand if it is needed	

GNOME Block Diagram



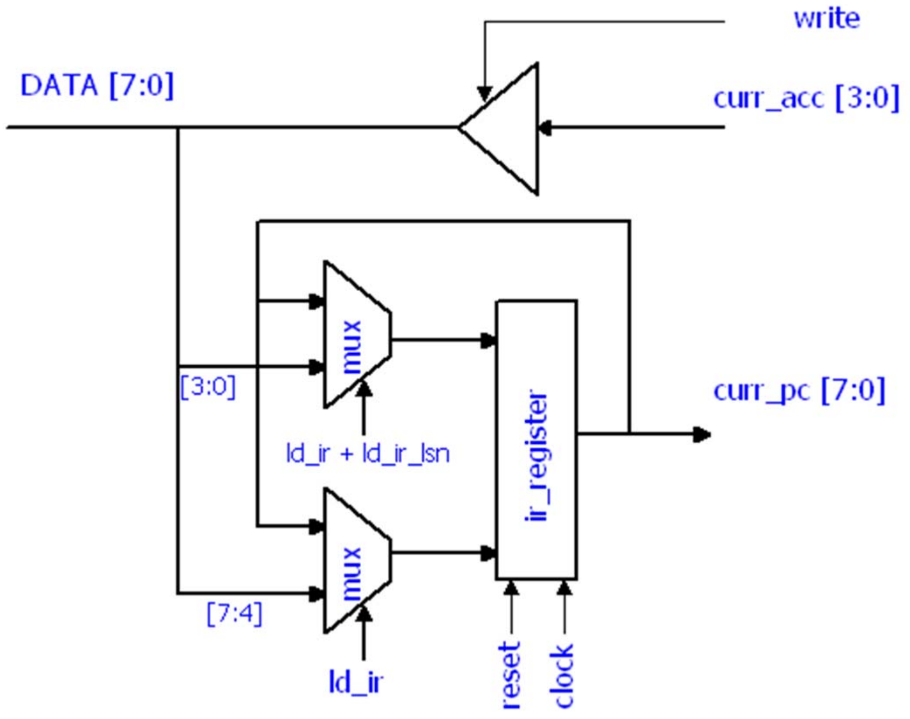
Address Generation

- ❑ If inc_pc = 1, the adder output goes to the input of the register (for normal instruction address update and skipz and skipc instructions).
- ❑ If jump_pc =1, curr_ir goes to the input of the register (for jump instruction)
- ❑ If jump_pc =0 and inc_pc=0, the output of the register goes to the input of the register.
- ❑ It is prohibited to have jump_pc =1 and inc_pc=1 at the same time.
- ❑ After reset, curr_pc=0. Thus, the microprocessor fetches the first instruction from address 000000

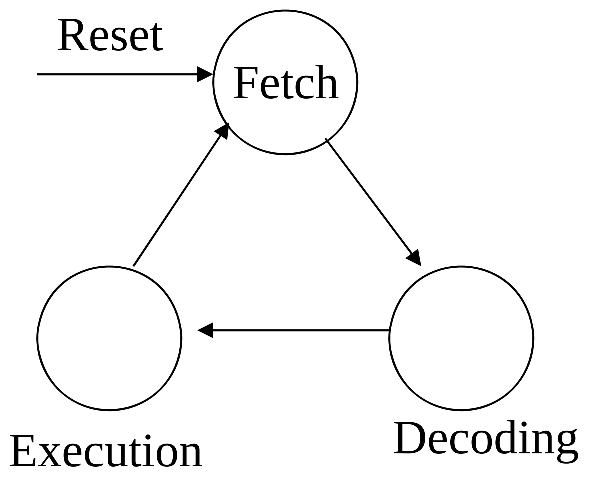


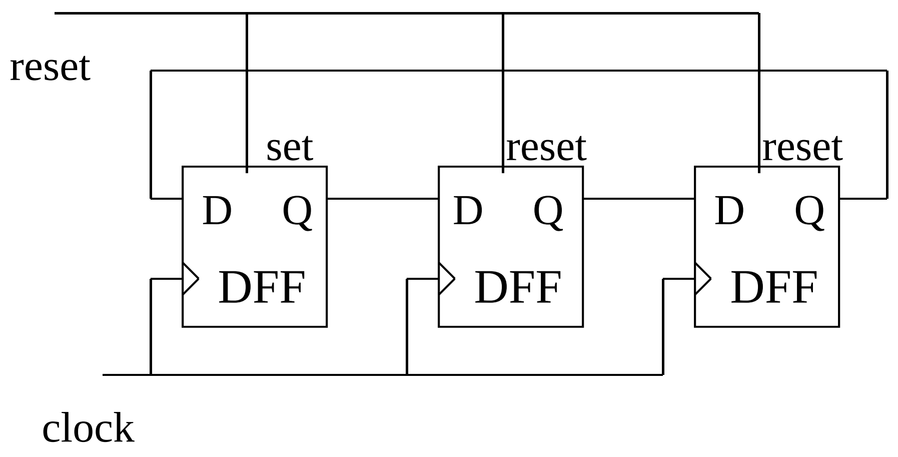
Instruction Latch & Data Bus Circuits

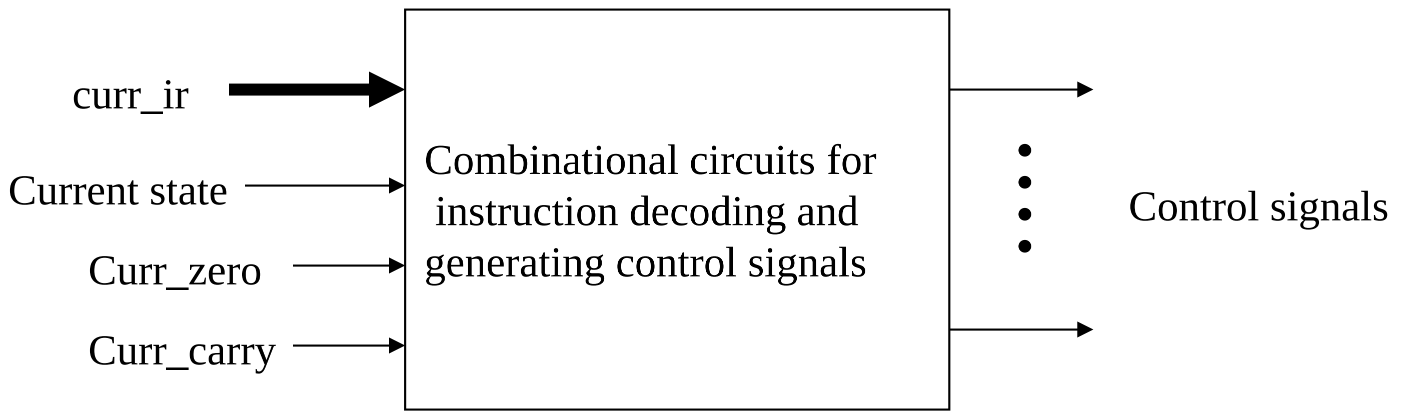
- ❑ For write operation (store rd), the write buffer is on and data on curr_acc are written into memory. During other time, the write buffer is off (high impedance output)
- ❑ To fetch an instruction, both muxs pass the data on the data bus to the instruction register
- ❑ To fetch data from data RAM, only the least significant mux passes the pass the data on the data bus to the instruction register



Control & decoding Unit






curr_ir
Current state
Curr_zero
Curr_carry
Combinational circuits for instruction decoding and generating control signals
Control signals

Control & decoding Unit

- Example: how generate control signal write

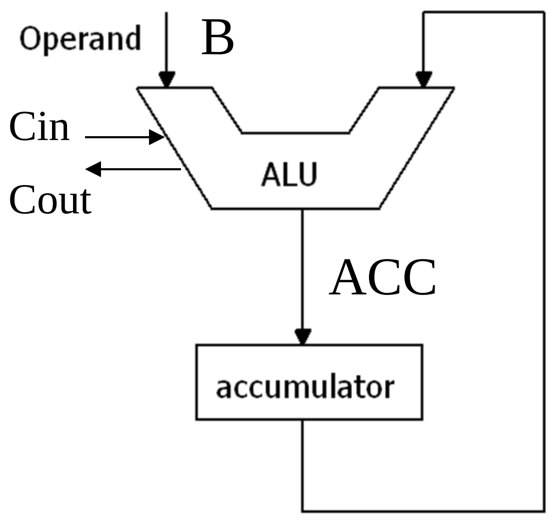
  — write signal should be driven to high to enable the write buffer during the execution state of ***store rd*** instructions

write = exe_state **AND** store_instruction

1. Signal exe_state is high when GNOME is during execution states
2. Signal store_instruction is high if the current instruction is ***store rd***

Execution Unit

- ❑ Accumulator-Based Architecture



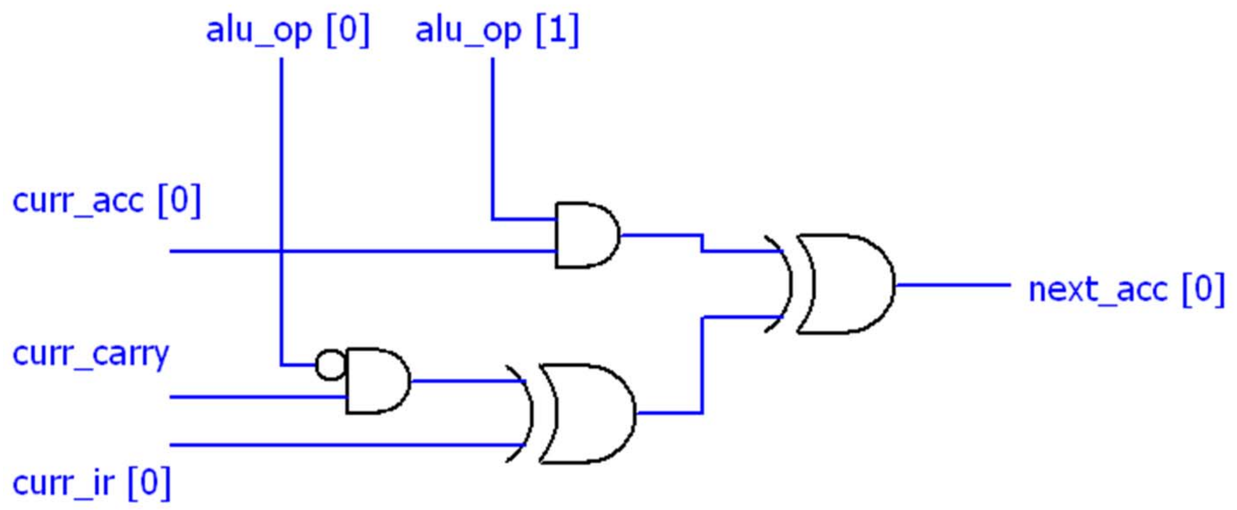
— For operations with two operands, one operand is accumulator.

— The execution result is store in accumulator

- ❑ ALU operations

Name	Operation	Encoding
PASS	ACC ← B	0 0 1
ADD	ACC←ACC+B+Cin	0 1 0
XOR	ACC ← ACC ⊗ B	0 1 1
AND	Zero flag = ACC • B	1 0 0
SET_C	Set carry flag	1 0 1
CLR_C	Clear carry flag	1 1 0

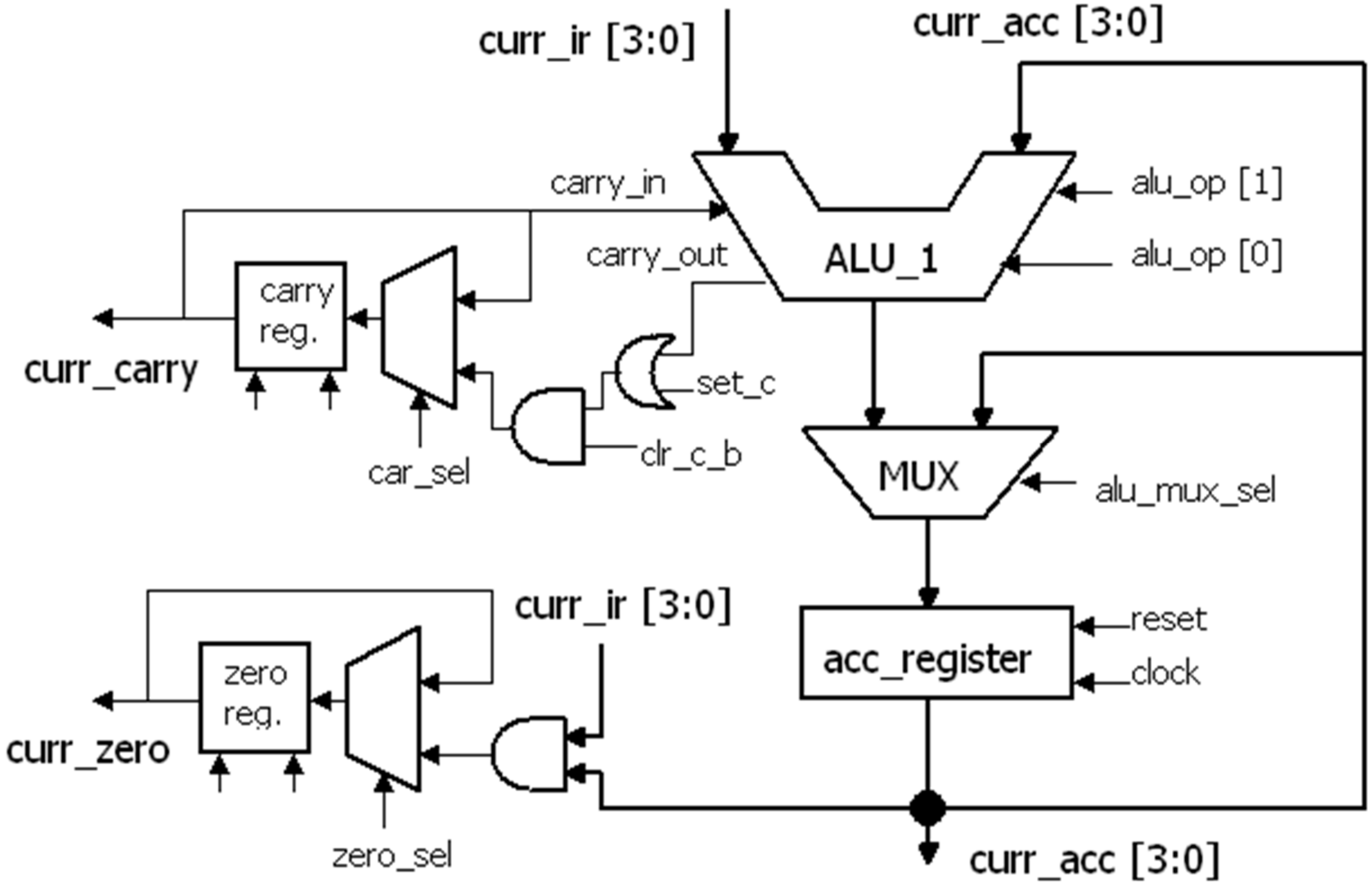
ALU Circuit



alu_op[1]	alu_op[0]	Operation
0	1	PASS
1	1	XOR
1	0	ADD*

* Extra circuits are needed for carry generation

Complete ALU Circuit

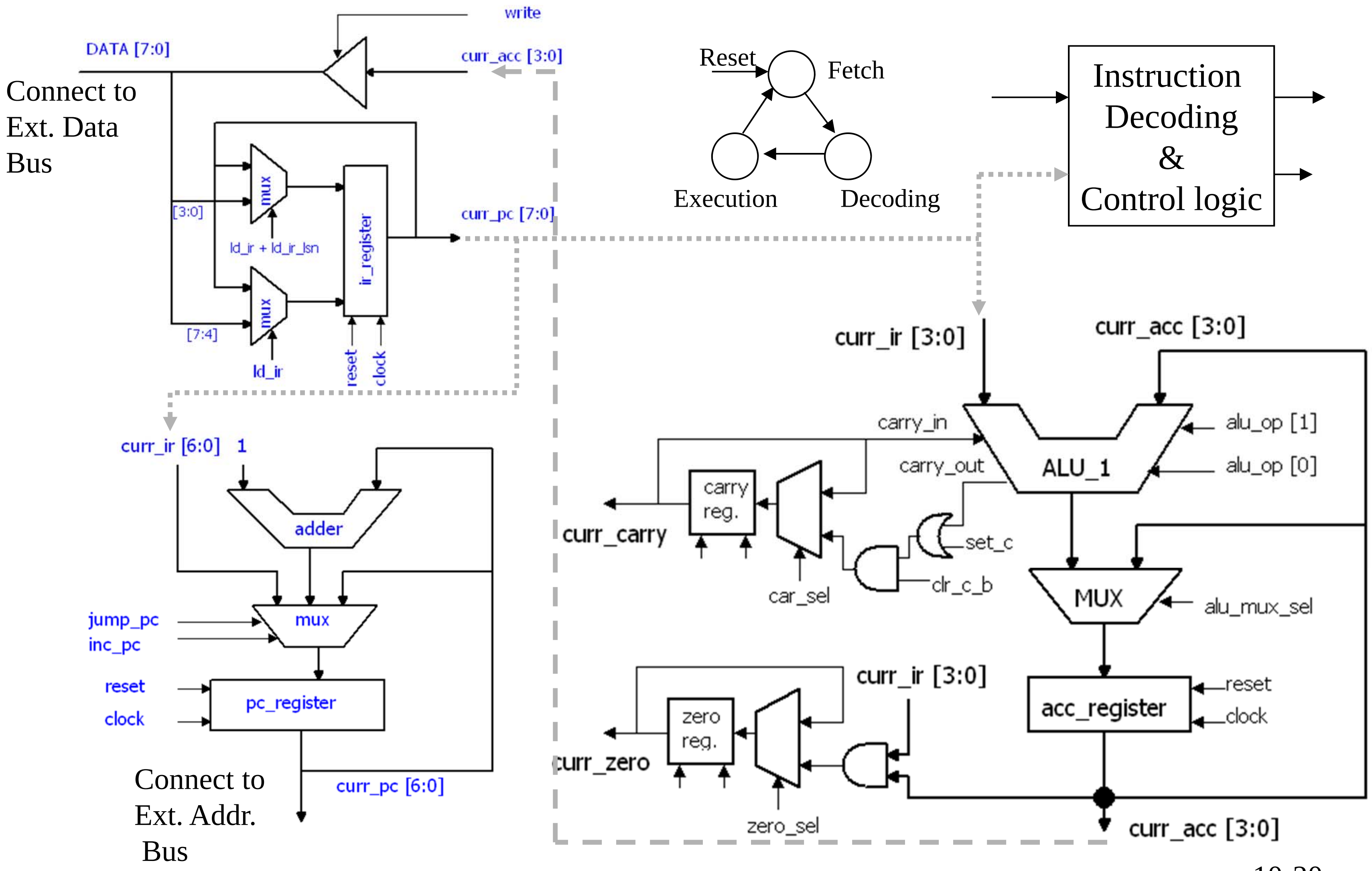


Programming GNOME Processor

- Example: writing a program to calculate 48H + 29H

Instructions		Machine code
Loadi	8	18H
Store	R0	30H
Loadi	4	14H
Store	R1	31H
Loadi	9	19H
Store	R2	32H
Loadi	2	12H
Store	R3	33H
Clear_c		00H
Load	R0	40H
Add	R2	52H
Store	R4	34H
Load	R1	41H
Add	R3	53H
Store	R5	35H

Put Everything Together



write
DATA [7:0]
curr_acc [3:0]
Connect to Ext. Data Bus
[3:0]
mux
ld_ir + ld_ir_lsn
ir_register
curr_pc [7:0]
[7:4]
mux
ld_ir
reset
clock
Reset
Fetch
Execution
Decoding
Instruction Decoding & Control logic
curr_ir [6:0]
1
adder
jump_pc
inc_pc
mux
reset
clock
pc_register
Connect to Ext. Addr. Bus
curr_pc [6:0]
curr_ir [3:0]
curr_acc [3:0]
carry_in
alu_op [1]
carry_out
ALU_1
alu_op [0]
carry reg.
curr_carry
set_c
clr_c_b
car_sel
MUX
alu_mux_sel
curr_ir [3:0]
acc_register
reset
clock
zero reg.
curr_zero
zero_sel
curr_acc [3:0]