CH7: Microarchitecture

System

Programming Language

ISA

Computer

Microarchitecture

Logic Gates

Transistors

How do we build a physical machine that can take physical representations of numbers that represent commands in the ISA

and perform the corresponding operations of moving physical representations of numbers between various physical storage units?

Overview

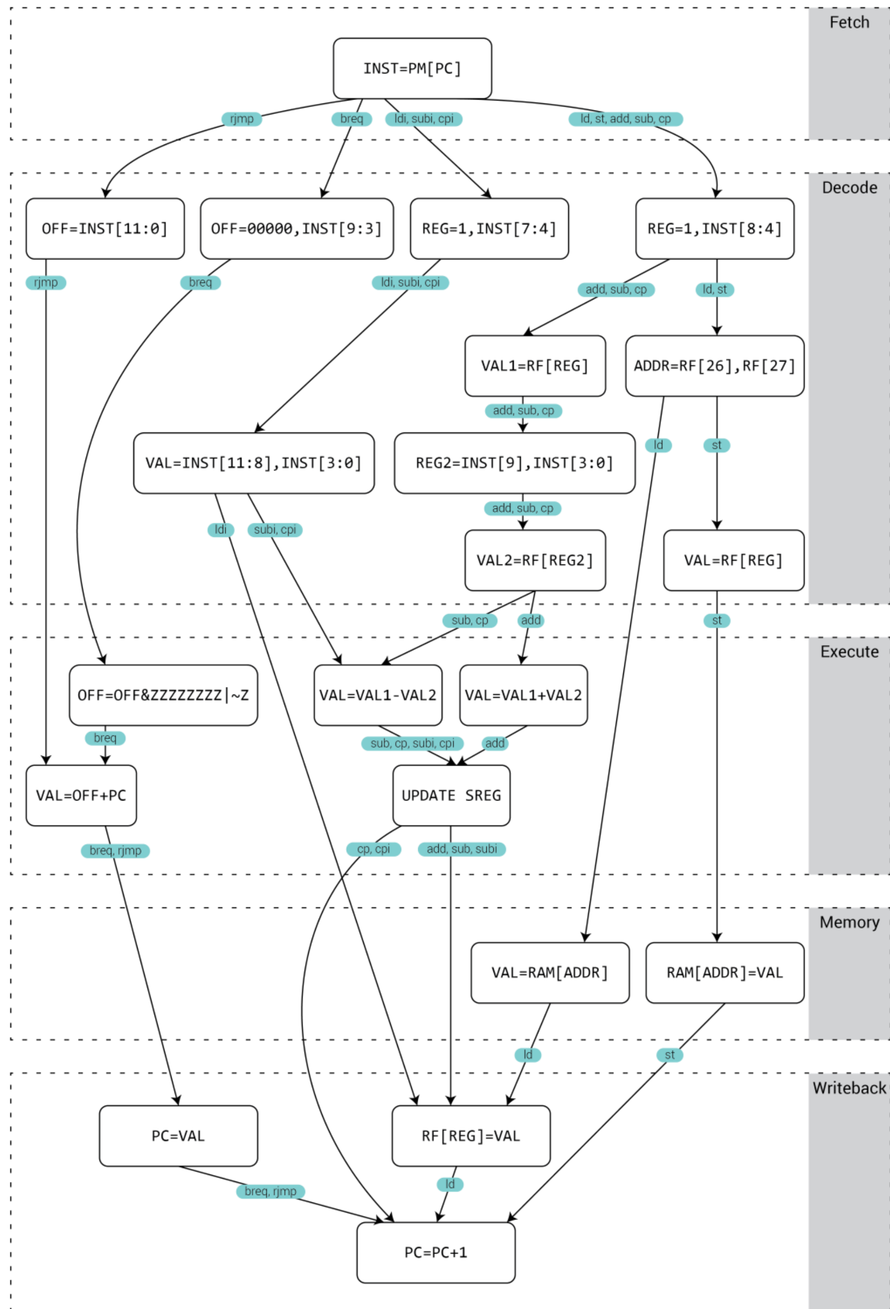
- ISA is an interface
  - Assembly language – text representation
  - Machine code – representation as numbers
- Microarchitecture
  - Implementation of the interface using logic gates (we will abstract away what exact logic gates are and deal with that in the next chapter)

Microarchitecture

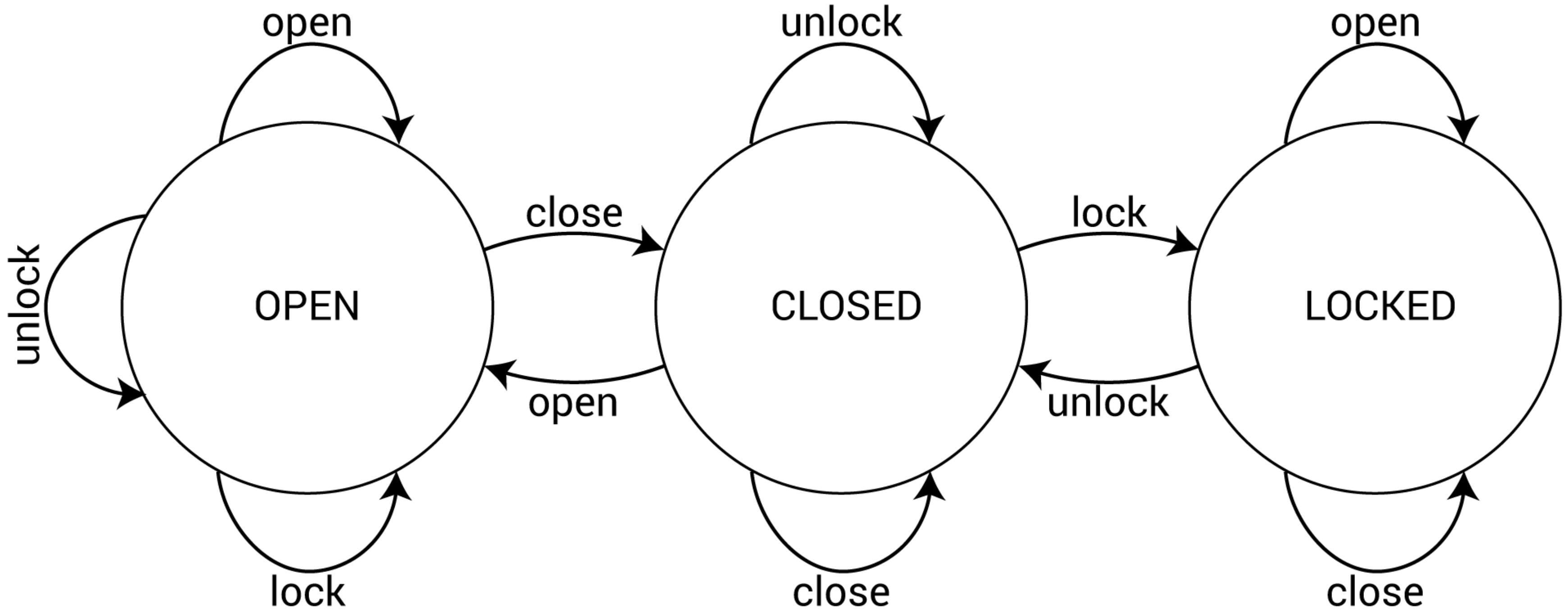
- Definition: Implementation of the ISA
- Two components
  - describe a computer as a specific kind of abstract machine called a **state machine**.
  - build this state machine as a physical device using various **circuit components**.

Today

- Concepts
  - State machine
  - Circuit component
- **LOTS** of new notation ☹


Fetch
INST=PM[PC]
rjmp
breq
ldi, subi, cpi
ld, st, add, sub, cp
Decode
OFF=INST[11:0]
OFF=00000,INST[9:3]
REG=1,INST[7:4]
REG=1,INST[8:4]
rjmp
breq
ldi, subi, cpi
add, sub, cp
ld, st
VAL1=RF[REG]
ADDR=RF[26],RF[27]
add, sub, cp
VAL=INST[11:8],INST[3:0]
REG2=INST[9],INST[3:0]
ld
st
add, sub, cp
ldi
subi, cpi
VAL2=RF[REG2]
VAL=RF[REG]
sub, cp
add
st
Execute
OFF=OFF&ZZZZZZZZ|~Z
VAL=VAL1-VAL2
VAL=VAL1+VAL2
breq
sub, cp, subi, cpi
add
VAL=OFF+PC
UPDATE SREG
breq, rjmp
cp, cpi
add, sub, subi
Memory
VAL=RAM[ADDR]
RAM[ADDR]=VAL
ld
st
Writeback
PC=VAL
RF[REG]=VAL
breq, rjmp
ld
PC=PC+1

State machine



State machine notation

- List of states (with “names”)
- Edges indicate transitions of when to go to next state

Microarchitecture state machine

- It is a simple 5 state state machine
  - Fetch an instruction
  - Decode it – figure out what values go where
  - Execute – perform computation
  - Memory – access memory if necessary
  - Writeback – write results somewhere

Fetch

- Remember that the value stored in the PC is the address of the current instruction
- In the fetch stage, we feed the value from PC into the program memory to read the actual instruction from that address.

Decode

- we need to figure out which instruction it is,
- and then what its various bits mean.
- For instance, if we get `1110010100011001`, we can decode this as an ldi instruction that should write value 01011001 to register 10001.

Execute

- Actual work happens here
- For e.g.: for add r20,r30, the decode stage will have determined that we're storing to r20, and will have read out the values from r20 and r30
- In the execute stage, then, we actually add the two values.
- We will also computing the new SREG

Memory

- Access memory if necessary to write or read
- Based on whether instruction is ld or st

Writeback

- Write the results to the register file

Auxiliary registers

- INST: For storing the current instruction, so of width 16
- REG: For storing a register number, so of width 5
- VAL: For storing something that would be stored in RAM or a register.

LDI

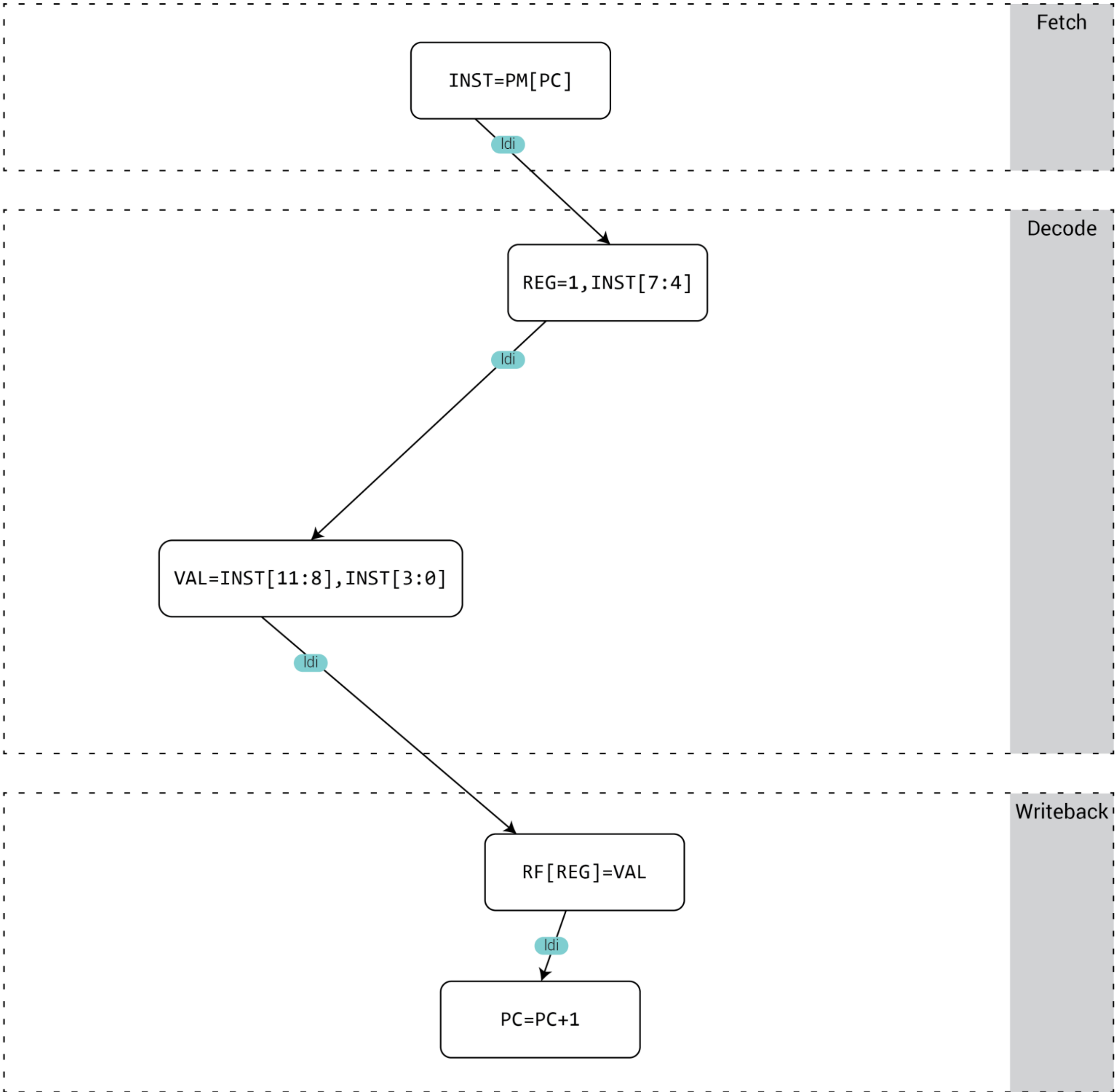
Instruction type	Format
4-bit register, 8-bit immediate	CCCCIIIIRRRRIIII

- LDI r17, 30
- *Fetch*: INST = PM[PC]
- *Decode:*
  - – REG = the bits of INST that say which register to write to
  - – VAL = the bits of INST that say which value to write
  - – **REG = 1,INST[7:4]**
  - – **VAL = INST[11:8],INST[3:0]**

LDI

Instruction type	Format
4-bit register, 8-bit immediate	CCCCIIIIRRRRIIII

- *Execute:* Nothing happens
- *Memory:* Nothing happens
- *Writeback:* write results
  - **RF[REG] = VAL**
  - PC = PC + 1

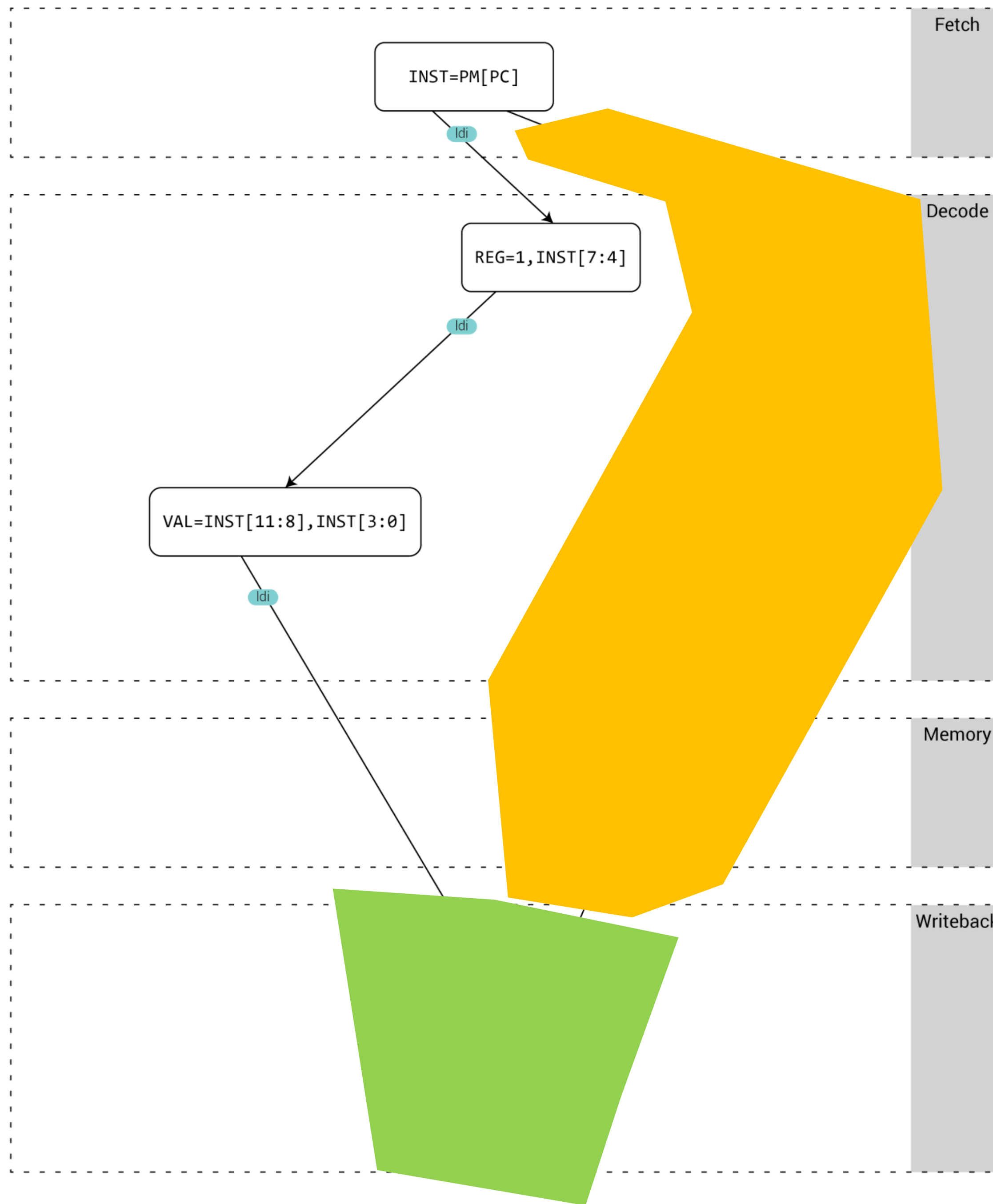

Fetch
INST=PM[PC]
ldi
Decode
REG=1,INST[7:4]
ldi
VAL=INST[11:8],INST[3:0]
ldi
Writeback
RF[REG]=VAL
ldi
PC=PC+1

LD

- *Fetch: same as LDI*
- *Decode:*
  - REG = bits of INST saying which register to write to
  - **ADDR** = bits of INST saying which address in data memory to read
  - **ADDR = RF[26],RF[27]**
  - *New auxilliary register*

LD

- Execute: nothing
- Memory
  - **VAL = data_memory[ADDR]**
- Writeback
  - **RF[REG] = VAL**



Fetch
INST=PM[PC]
ldi
Decode
REG=1,INST[7:4]
ldi
VAL=INST[11:8],INST[3:0]
ldi
Memory
Writeback

We can do this for each instruction ☺