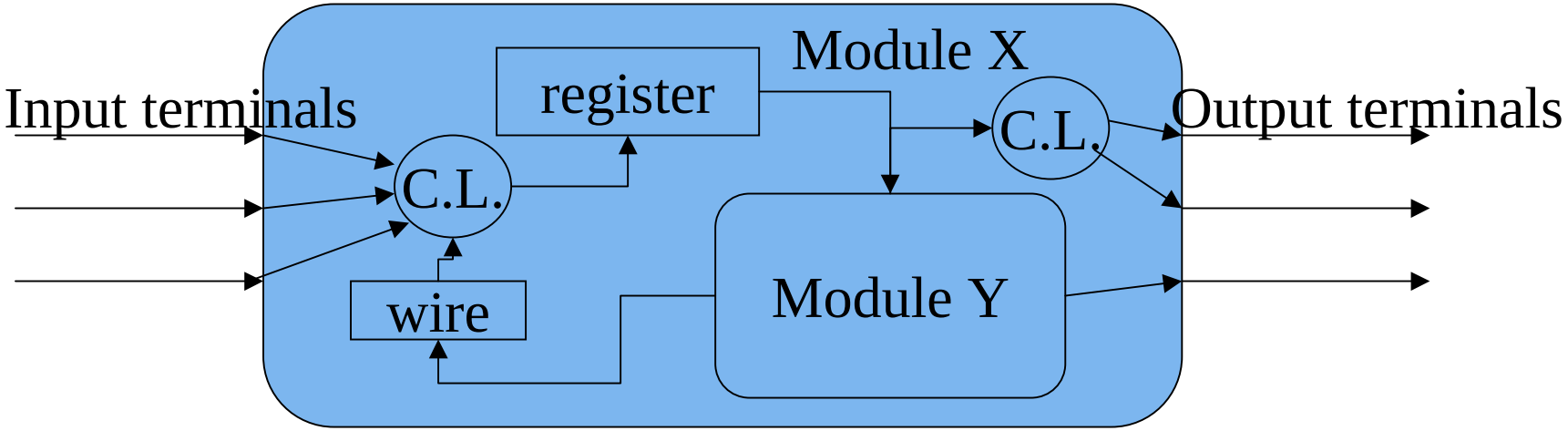
Behavior CPU design with NSL

Naohiko Shimizu, Ph.D.

You will have detailed information for NSL at http://www.overtone.co.jp/

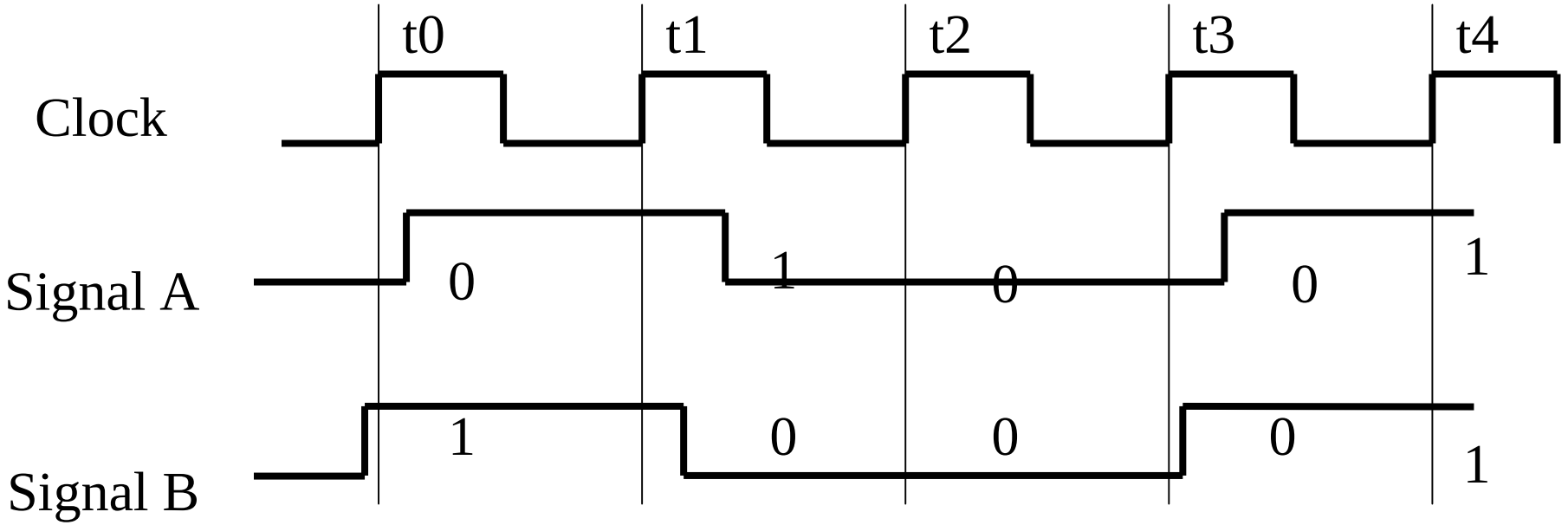
Concept

- A hardware is consist of registers, terminals, modules.
  - The design designates the behavior of the hardware which consists of timing or condition and values for registers and/or terminals.



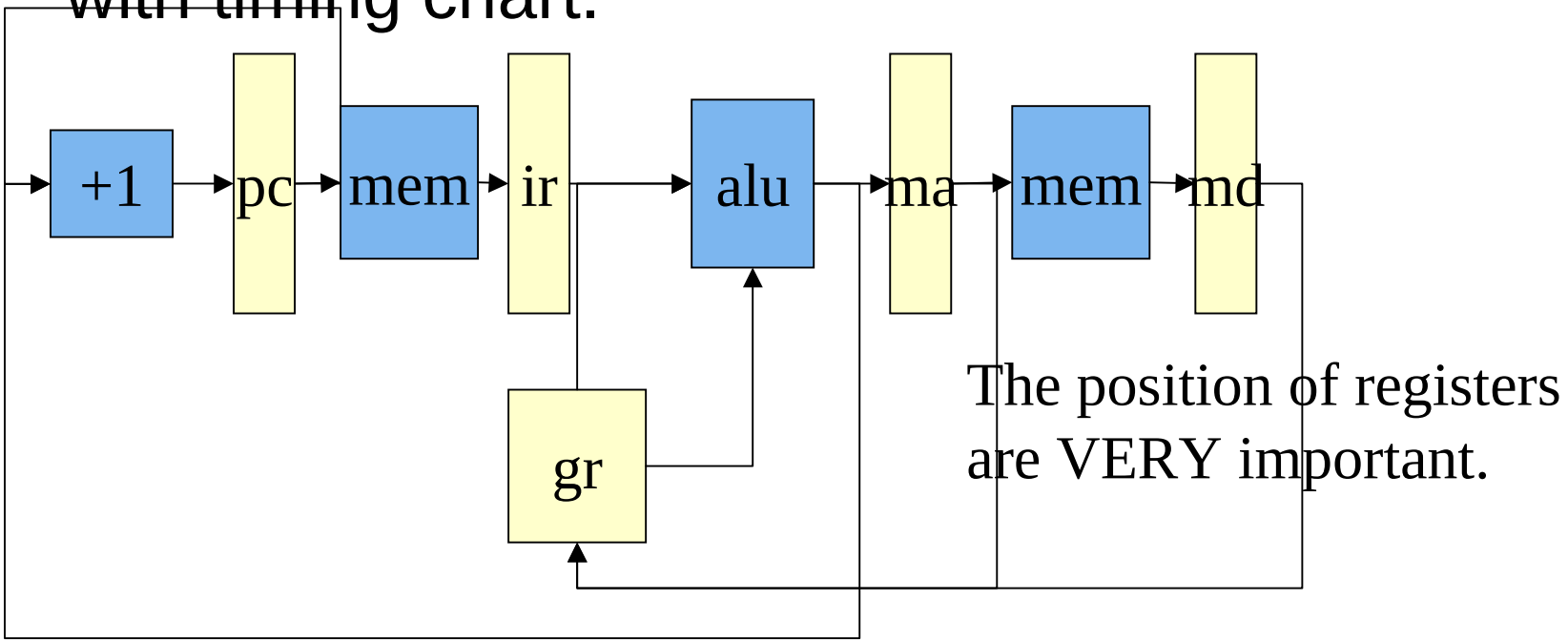
Clock synchronized design

- We treat all the signal transitions between clock phase as simultaneous signals at the rising edge of clock.



Data path design

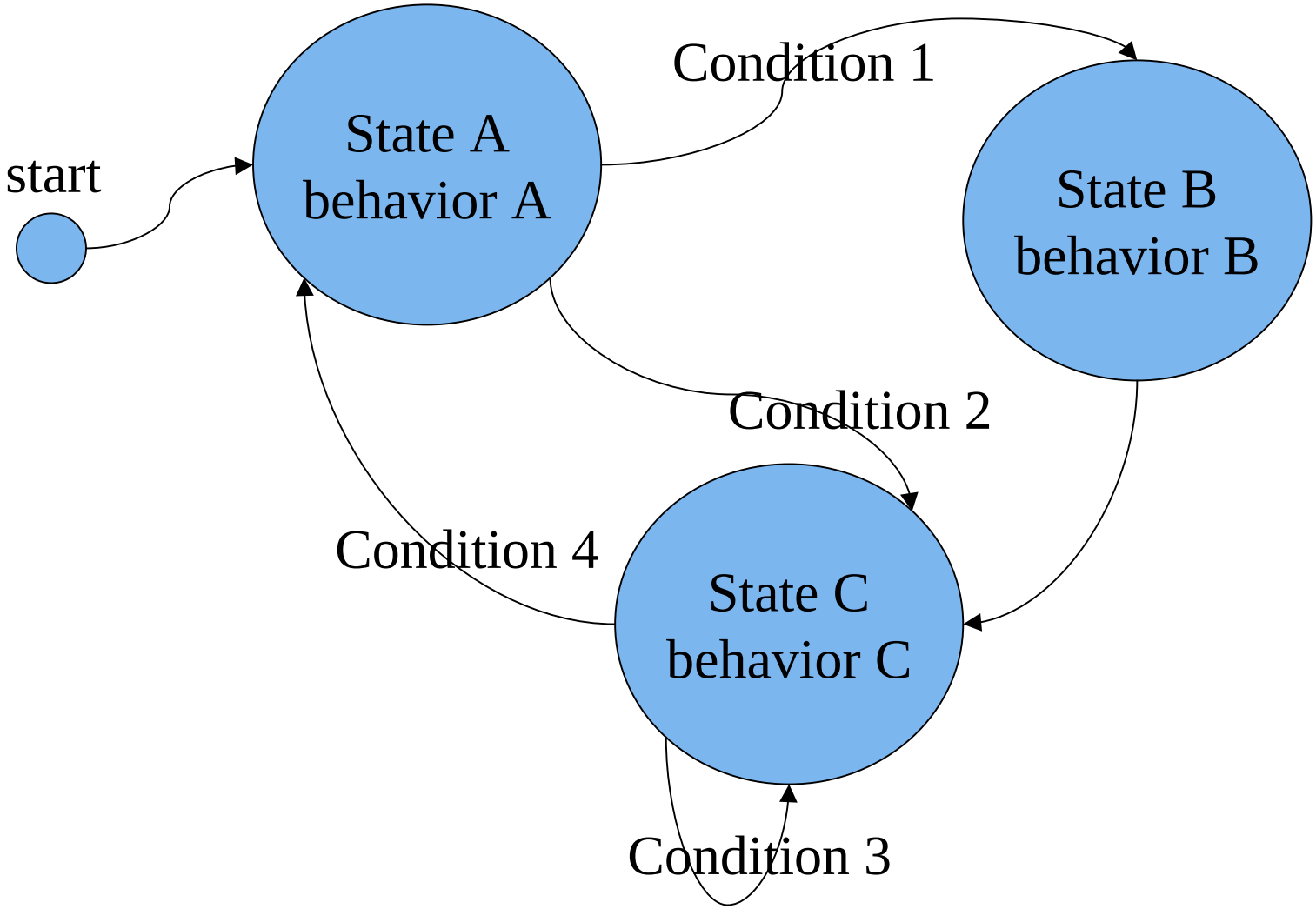
- Data path most affects on performance, chip size, power consumption.
  - Design the data path and check the data flow with timing chart.



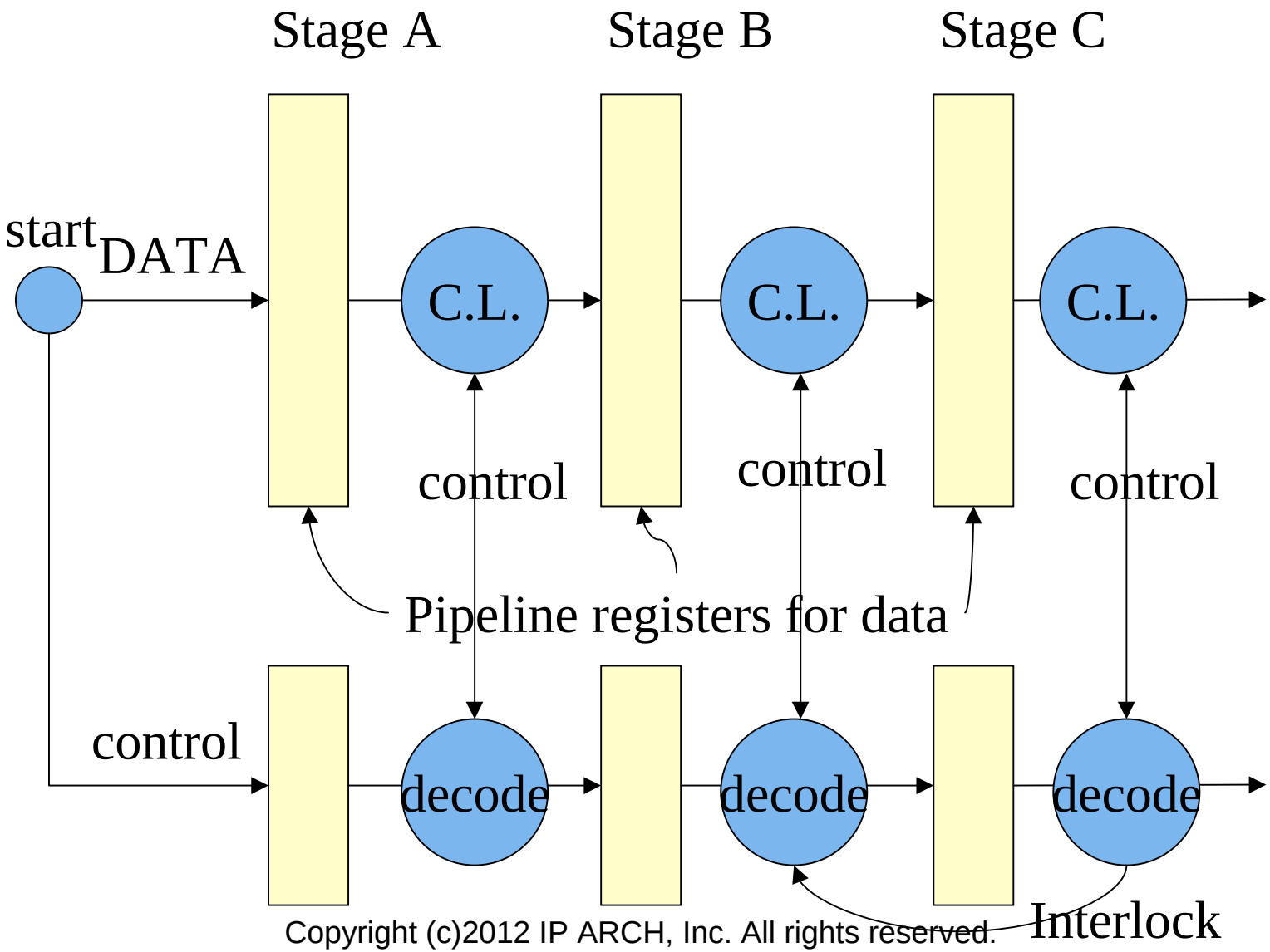
Hardware algorithm design

- Hardware designer prefers two methods to describe the hardware algorithm.
  - State Machine
    - behavior circles and arrows with transition condition
  - Pipeline control
    - consist of number of stages
    - Control commands flow into the stages with data in pipeline

State Machine



Pipeline

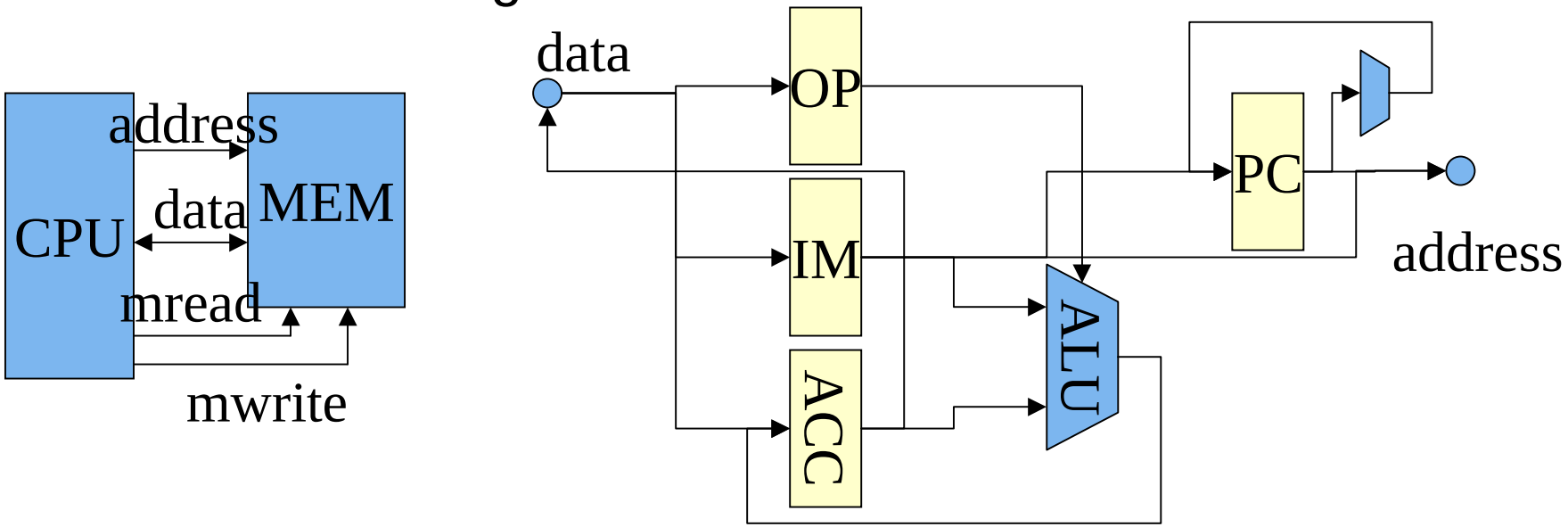


HDL design

- List up required components such as registers, wires, terminals.
- Design state machine.
- Write the behavior of each state.

Example design: CPU

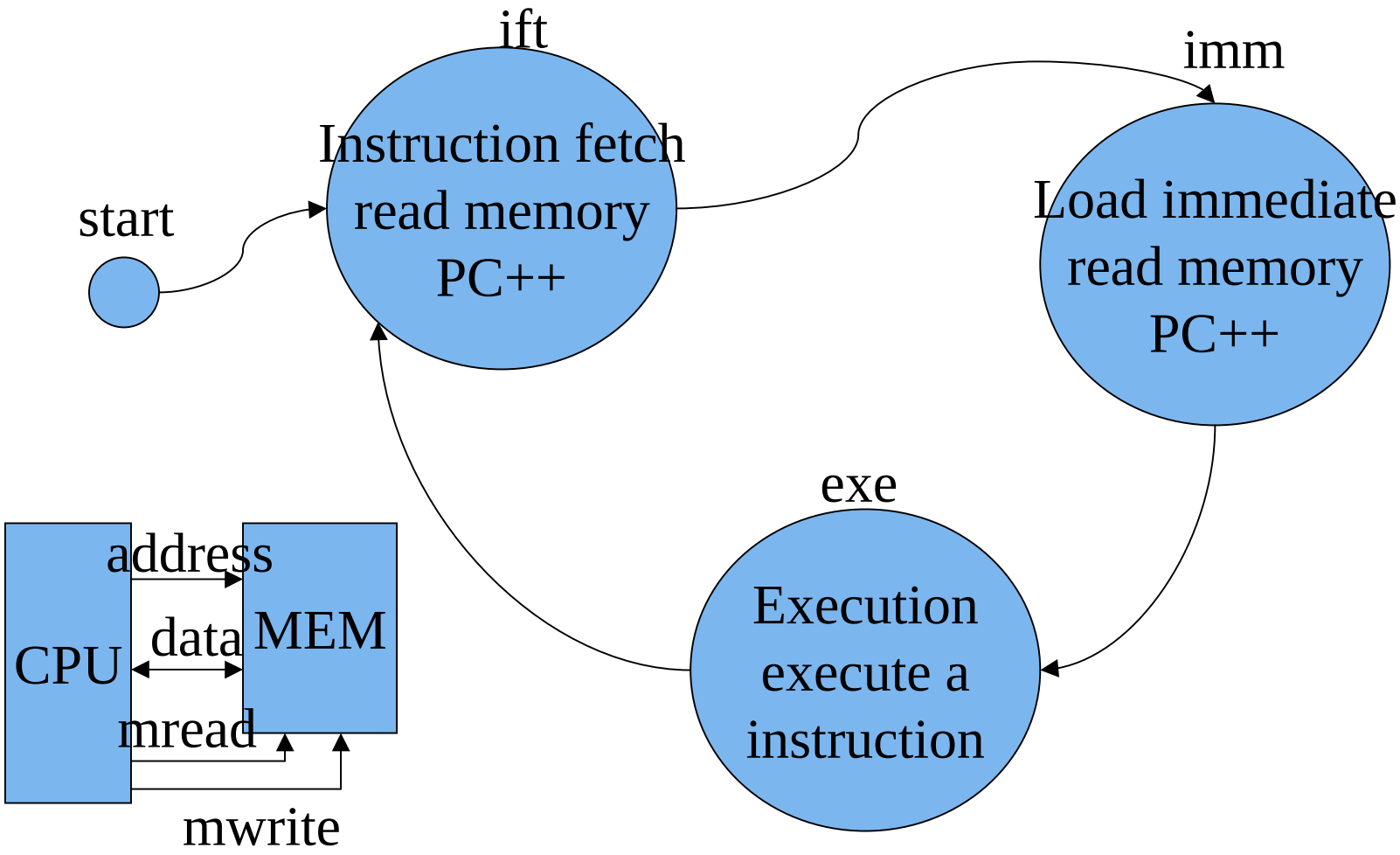
- Make a small CPU of your own.
  - Software visible register: PC, ACC all are 8bit registers.
  - Internal register OP:holds operational code
  - internal register IM: holds immediate value



Instructions

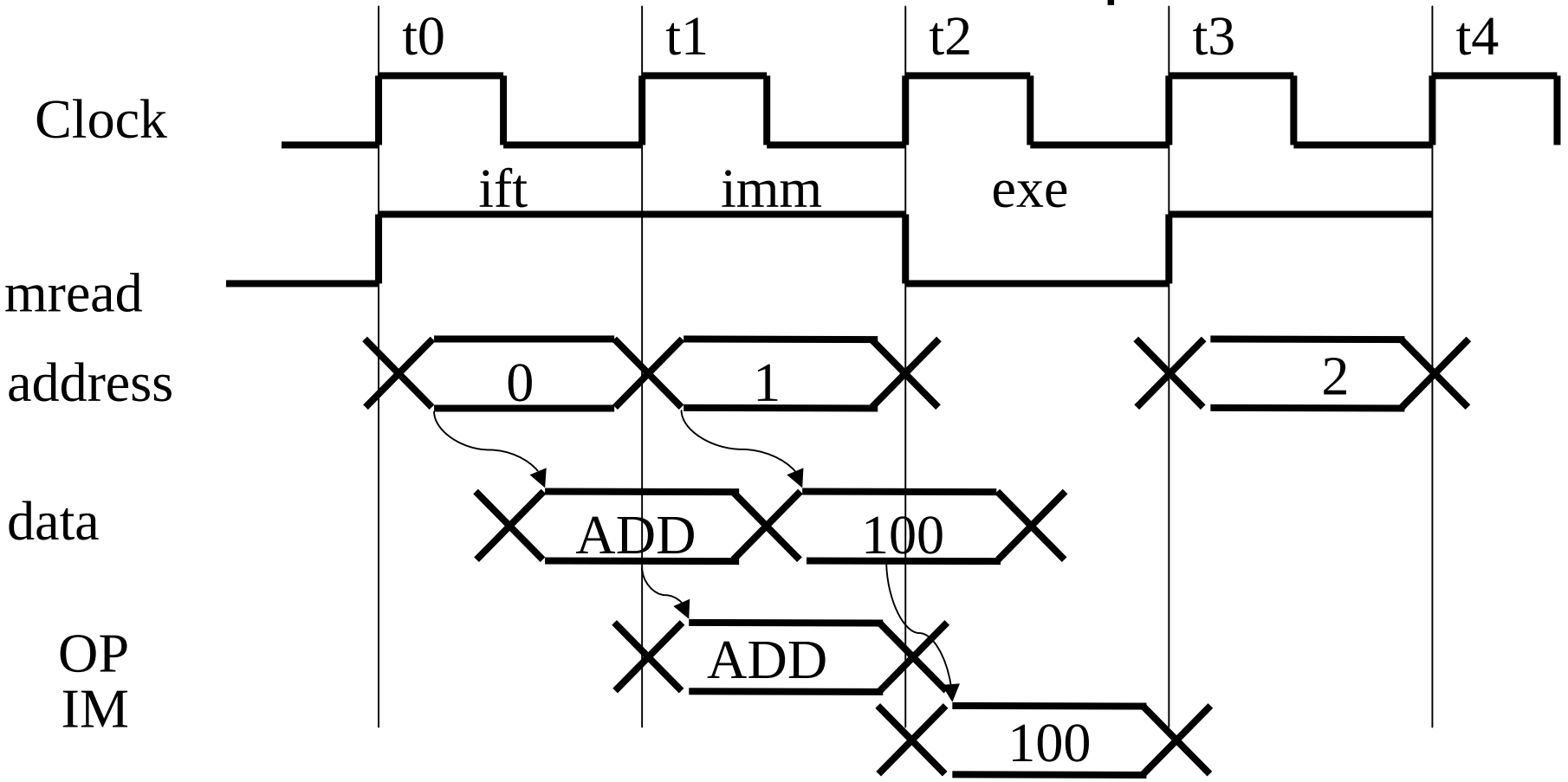
- List of instructions
  - ADD: add memory to accumulator
  - LI: load an immediate to accumulator
  - LD: load from memory to accumulator
  - ST: store to memory from accumulator
  - JZ: jump if accumulator is equal to zero
  - JMP: unconditional jump

State machine of CPU



Timing chart

- Instructions takes 3 clocks to operate



Writing NSL: part1

```
#define ADD 0
#define LD  1
#define ST  2
#define JMP 3
#define JZ  4
#define LI 5
declare cpu {
    input datai[8];
    output datao[8];
    output address[8];
    func_out mread(address) : datai;
    func_out mwrite(address,datao);
}
```

OP code defines

data buses

Address output bus

Memory reading control
send with address,
result will available in data

Memory write control
send with address, data

Writing NSL: part2

```
module cpu {
  reg count[4]=0, pc[8], op[8], im[8],
 acc[8]=0;
  proc_name ift(pc), imm(op), exe(im);
```

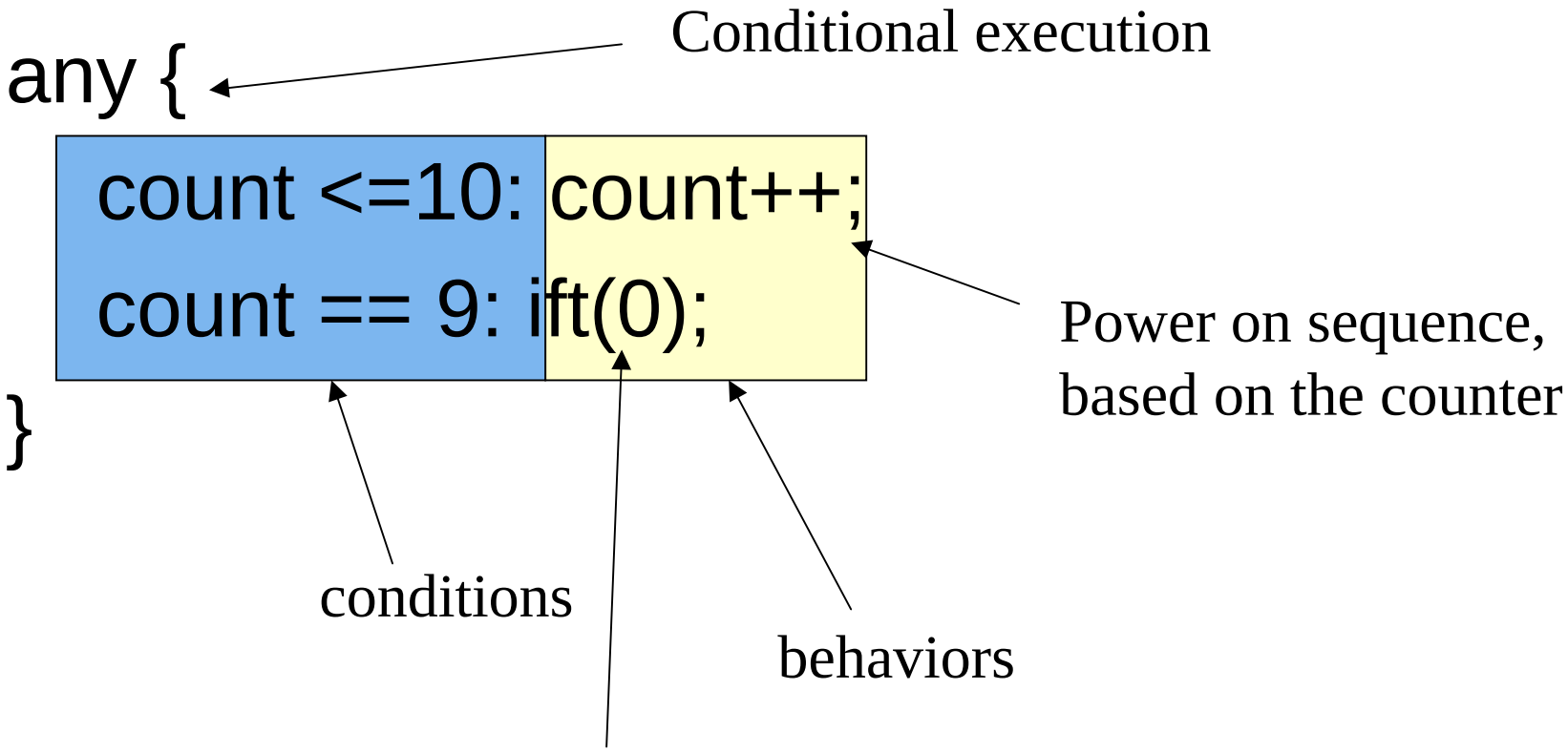
Module name is cpu

Power on initialize to 0

Define internal resources.
Registers are defined.

Define internal control procedures
which corresponds to states.
Each procedures may have register
parameters.

Writing NSL: part3



Calling ift procedure with argument 0
which means start instruction of address 0.
Argument will transfer to parameter PC.

Writing NSL: part4

```
proc ift {
    imm(mread(pc++));
}

proc imm {
    exe(mread(pc++));
}
```

Procedure ift behavior definition.

Call procedure IMM
with argument of data.
The mread control signal's
argument is PC, and the
memory read data is supplied
to parameter OP.
After then, PC will be increased.

Similar procedure IMM will call EXE.
The memory data is set to parameter IM.

Writing NSL: part5

```
proc exe {
    wire nextpc[8];
    any {
        op == ADD: acc:=acc+mread(im);
        op == LD:  acc:=mread(im);
        op == ST:  mwrite(im,acc);
        op == LI:  acc:=im;
    }
    any {
        op == JMP: nextpc=im;
        (op == JZ) && (acc == 0): nextpc=im;
    else: nextpc=pc;
    }
    ift(nextpc);
}   }
```

Procedure exe behavior definition.

Temporary wire terminal.

Operation's for usual inst.

Set next instruction address.

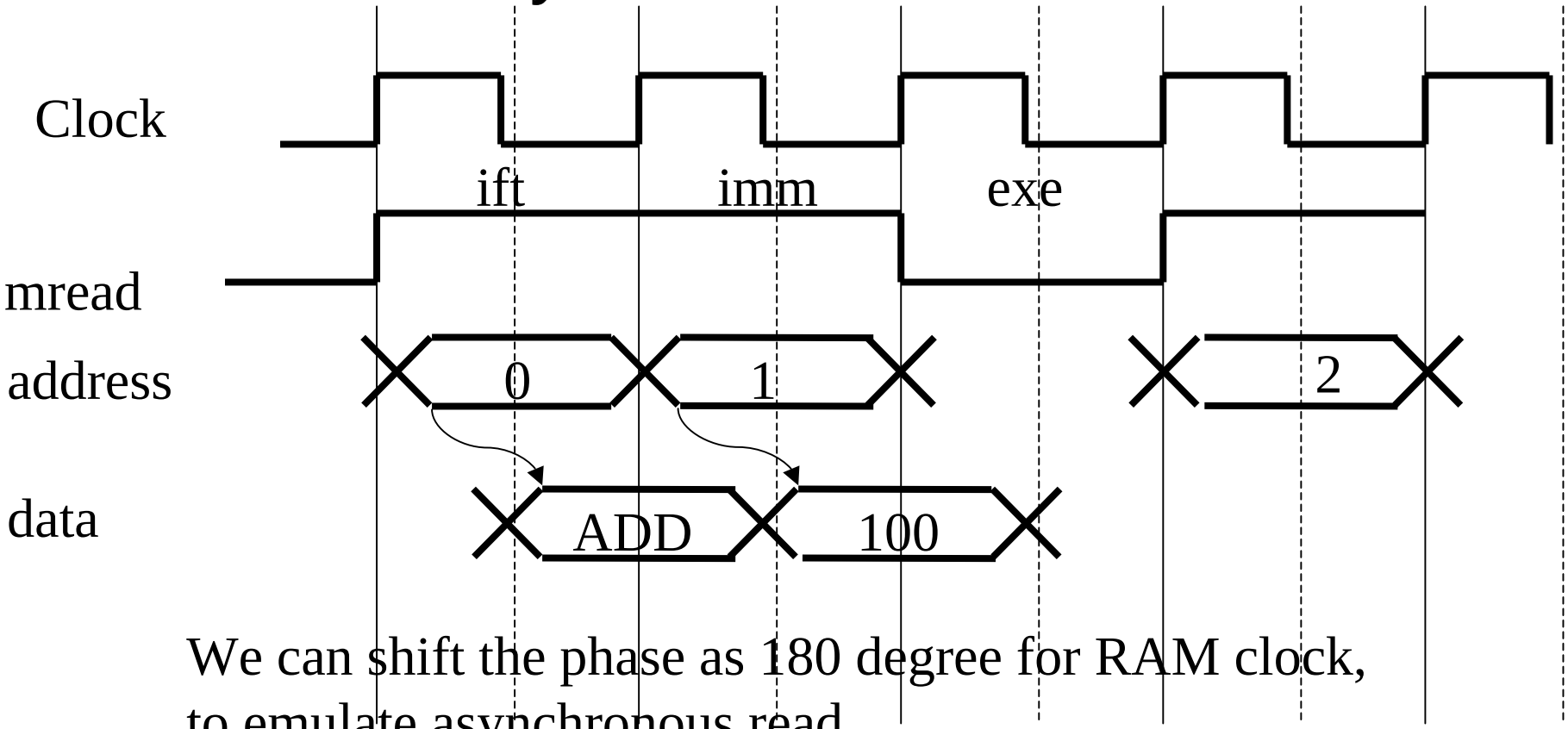
Call instruction fetch procedure.

Putting it on FPGA

- Memory model adjustment
  - NSL uses synchronous write, asynchronous read memory.
  - You may need to insert glue logic for adjusting setup and/or hold time.
- pin assignment
  - We need to assign the physical numbers of designed pins and power/reset pins.

Synchronous RAM

- FPGA uses synchronous internal rams.



We can shift the phase as 180 degree for RAM clock,
to emulate asynchronous read.

Synchronous ram modeling

```
declare ram256 {
    inout      data [8] ;
    input      adr[8] ;
    func_in    rd(adr):data
    func_in    wt(adr,datai);
}
module ram256 {
    reg dout[8];
    mem ram[256][8]={3,0};
    dout := ram[adr];
    func rd   data = dout;
    func wt  ram[adr] := data;
}
```

See actual cpu.nsl file for current implementation.

Initial value of memory.
Jump to 0 (infinite loop)

Registered output data

Top module

- We will have memory and CPU.

```
declare top simulation {}
module top {
        cpu mycpu;
        ram256 mainmem;

        mainmem.m_clock = ~m_clock;

        func mycpu.mread mycpu.data = mainmem.rd(mycpu.address);
        func mycpu.mwrite mainmem.wt(mycpu.address, mycpu.data);
 _init {
                _delay(20);
                _finish();
        }
}
```

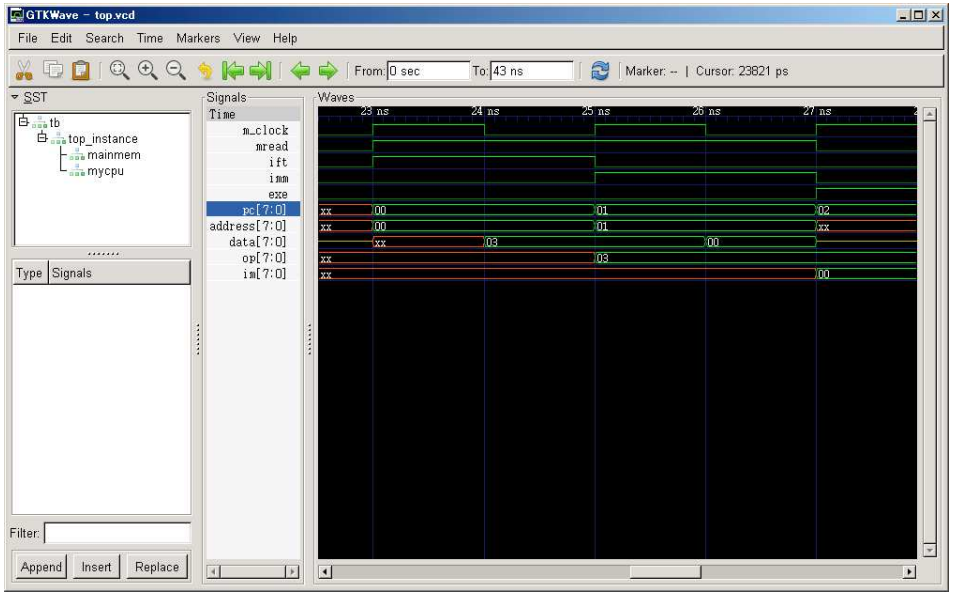
See actual cpu.nsl file for current implementation.

Phase shift 180 degree.

After 20 clocks, stop simulation.

Run simulation

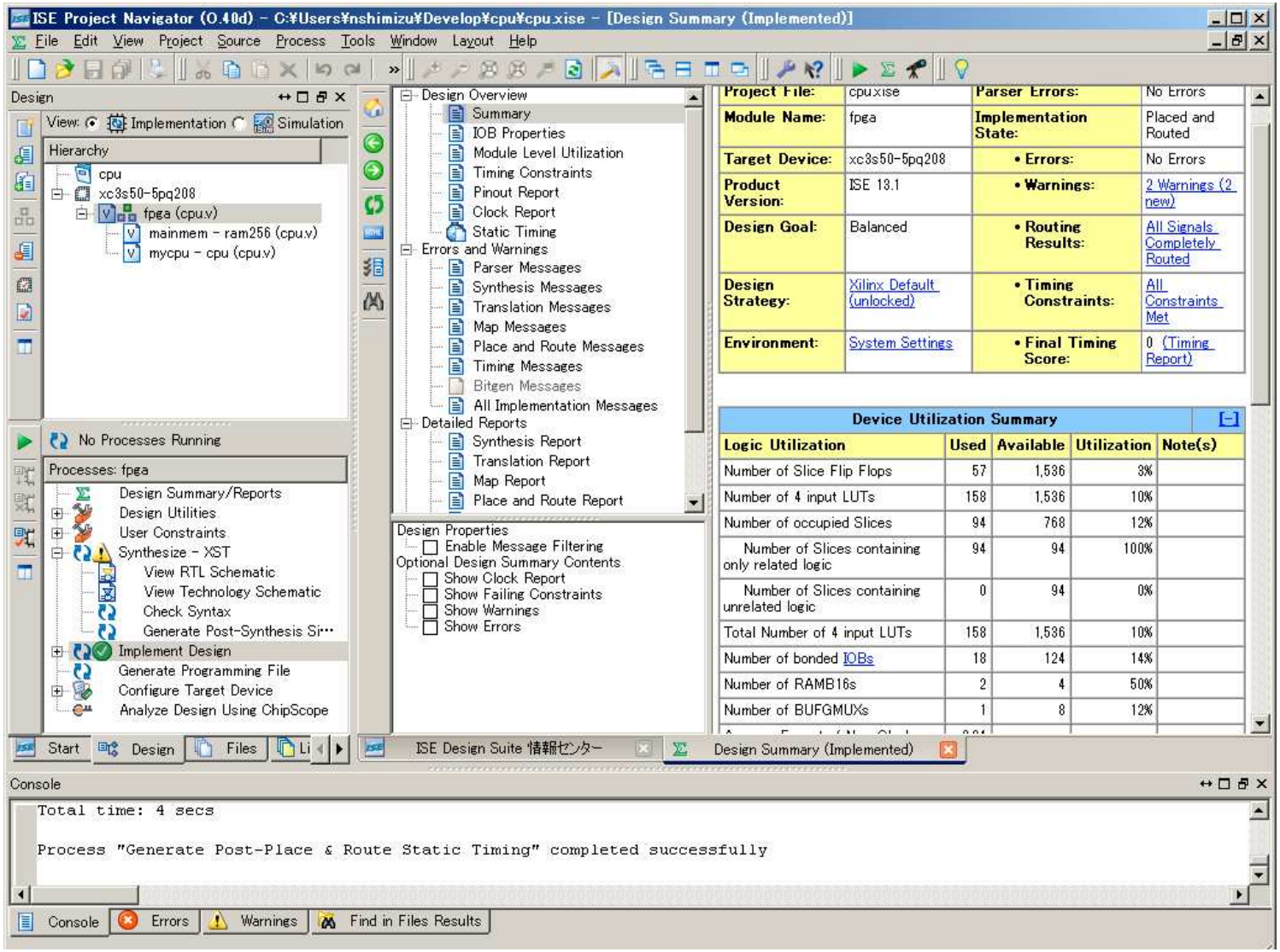
- Compile and run
  - nsl2vl cpu.nsl -verisim2 -target top
  - iverilog cpu.v
  - vvp a.out
  - gtkwave top.vcd cpu.sig



Concept

- In NSL, designer describes the behavior which is the set of conditions and values.
  - NSLCORE analyzes the behavior and decompose it for registeres/terminals/modules. . Then it generates RTL.
- Unlike C language, designer can explicitly designate the registers which is the key components of hardwares.

Fitting result on FPGA



Conclusion

- You got a CPU with NSL.
  - It works on simulator and/or FPGA.
- For advanced students, you can modify the CPU and make useful instructions for your project.