Lecture 07 08 09
Pipeline Processor with Hazard

Jun WANG

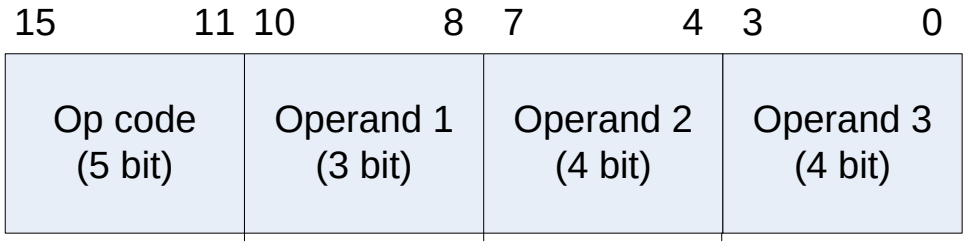
Contents

- ISA (instruction set architecture)
  - Operation field
  - Storage and access
  - Operations
- MA (micro architecture)
  - Top-view and block diagram
  - Verilog for stages
- Hazard in MA

Design target

- RISC CPU
- Data path 16b
- Data memory
  - 2^8 X 16b
- Operation memory:
  - 2^8 X 16b
  - Size of operation set: 2^5
- General register
  - 8 X 16b
- Flags
  - NF, ZF, CF
- Control
  - Clock, reset, enable, start
- Testing
  - 4 bit selection for 16 bit output

Operation field



R type (register type)	r1 r1	r2 r2	r3 -
I type (Immediate type)	r1 -	val2 val2	val3 val3
RI type	r1	r2	val3

r1
000: gr[0]
001: gr[1]
010: gr[2]
011: gr[3]
100: gr[4]
101: gr[5]
110: gr[6]
111: gr[7]

r2
x000: gr[0]
x001: gr[1]
x010: gr[2]
x011: gr[3]
x100: gr[4]
x101: gr[5]
x110: gr[6]
x111: gr[7]

val2
Immediate data (4 bit)

r3
x000: gr[0]
x001: gr[1]
x010: gr[2]
x011: gr[3]
x100: gr[4]
x101: gr[5]
x110: gr[6]
x111: gr[7]

val3
Immediate data (4 bit)

gr: general register
(16 bit X 8)

Access memory and register

- Access register:
  - Ex. gr[r1],
  - r1 for simplification
  - [r1]/r1gets register number
- Access memory:
  - Ex. m[r2+val3],
  - [r2+val3]/[gr(r2)+val3] gets address of memory
- Access immediate data:
  - Ex. {val2, val3},
  - MSB: val2, LSB:val3

An example of operation codes (assemble language)

- LOAD gr1, gr0, 0
- LOAD gr2, gr0, 1
- NOP
- NOP
- NOP
- ADD gr3, gr1, gr2
- NOP
- NOP
- NOP
- STORE gr3, gr0, 2
- HALT

In C code:
Y = A + B
M[gr0+2] =M[gr0+1]+M[gr0+0]

Operation

- Data transfer & Arithmetic

mnemonic	operand1	operand2	operand3	op code	operation
NOP *				00000	no operation
HALT *				00001	halt
LOAD *	r1	r2	val3	00010	gr[r1]<-m[r2+val3]
STORE *	r1	r2	val3	00011	m[r2+val3]<-r1
LDIH	r1	val2	val3	10000	r1<-r1+{val2, val3, 0000_0000} (lower 8'b0 can be given with ADDI)
ADD *	r1	r2	r3	01000	r1<-r2+r3
ADDI	r1	val2	val3	01001	r1<-r1+{val2, val3}
ADDC	r1	r2	r3	10001	r1<-r2+r3+CF
SUB	…				
SUBI	…				
SUBC	…				
CMP *		r2	r3	01100	r2-r3; set CF,ZF and NF

*: operations have been implemented in the following verilog

Operation

- Logical / shift

mnemonic	operand 1	operand2	operand3	op code	operation
AND	r1	r2	r3	01101	r1<-r2 and r3
OR	…				
XOR	…				
SLL	r1	r2	val3	00100	r1<-r2 shift left logical (val3 bit shift)
SRL	…				
SLA	r1	r2	val3	00101	r1<-r2 shift left arithmetical (val3 bit shift)
SRA	…				

Logical shift: 1001 SRL 2b → 0010 (append 0)

Arithmetical shift: 1001 SRA 2b → 1110 (keep the sign)

Operation

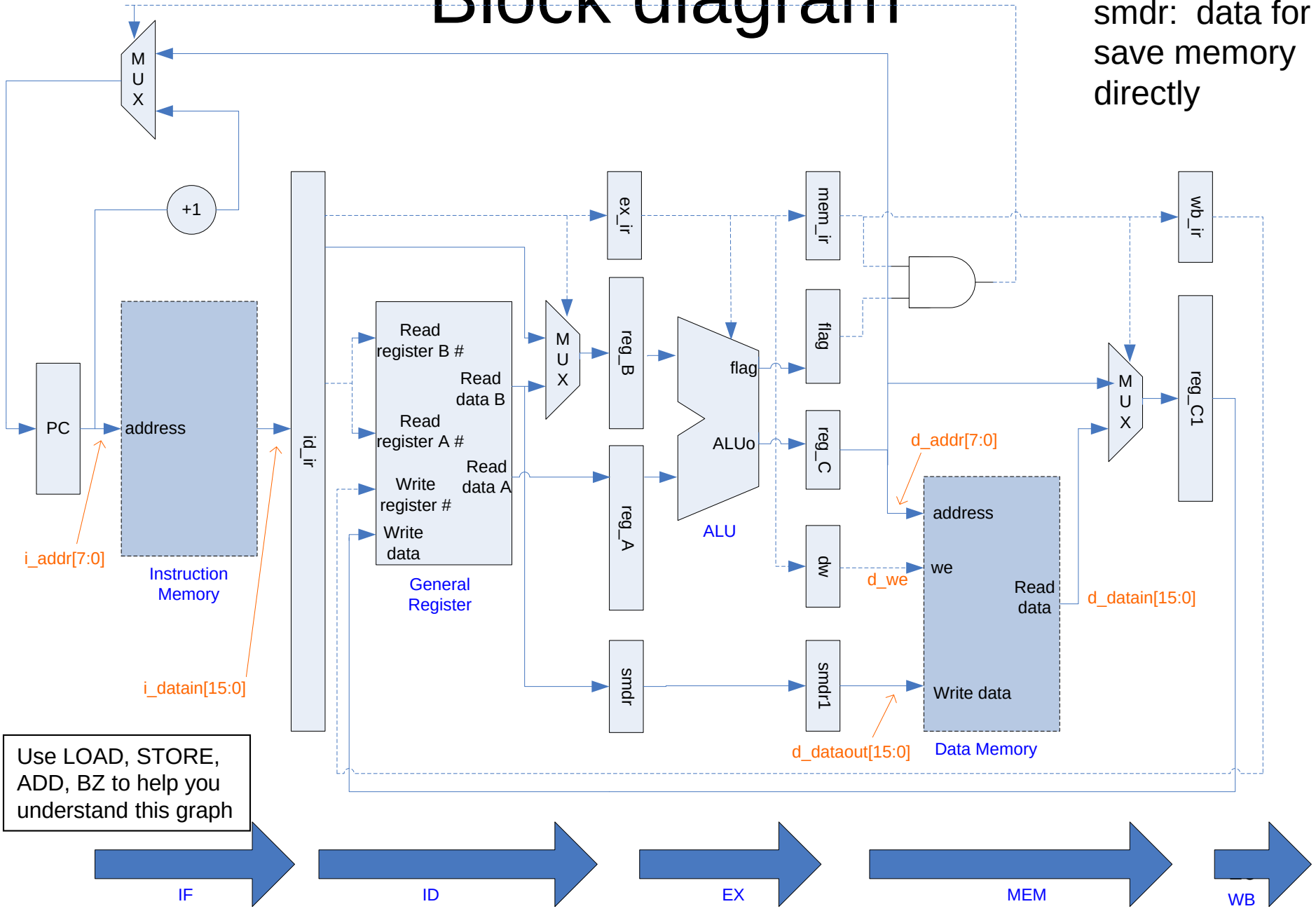
- Control

Flag registers:
-Used for control operations (conditional branch)
-ZF (zero flag), NF (negative flag), CF (carry flag)

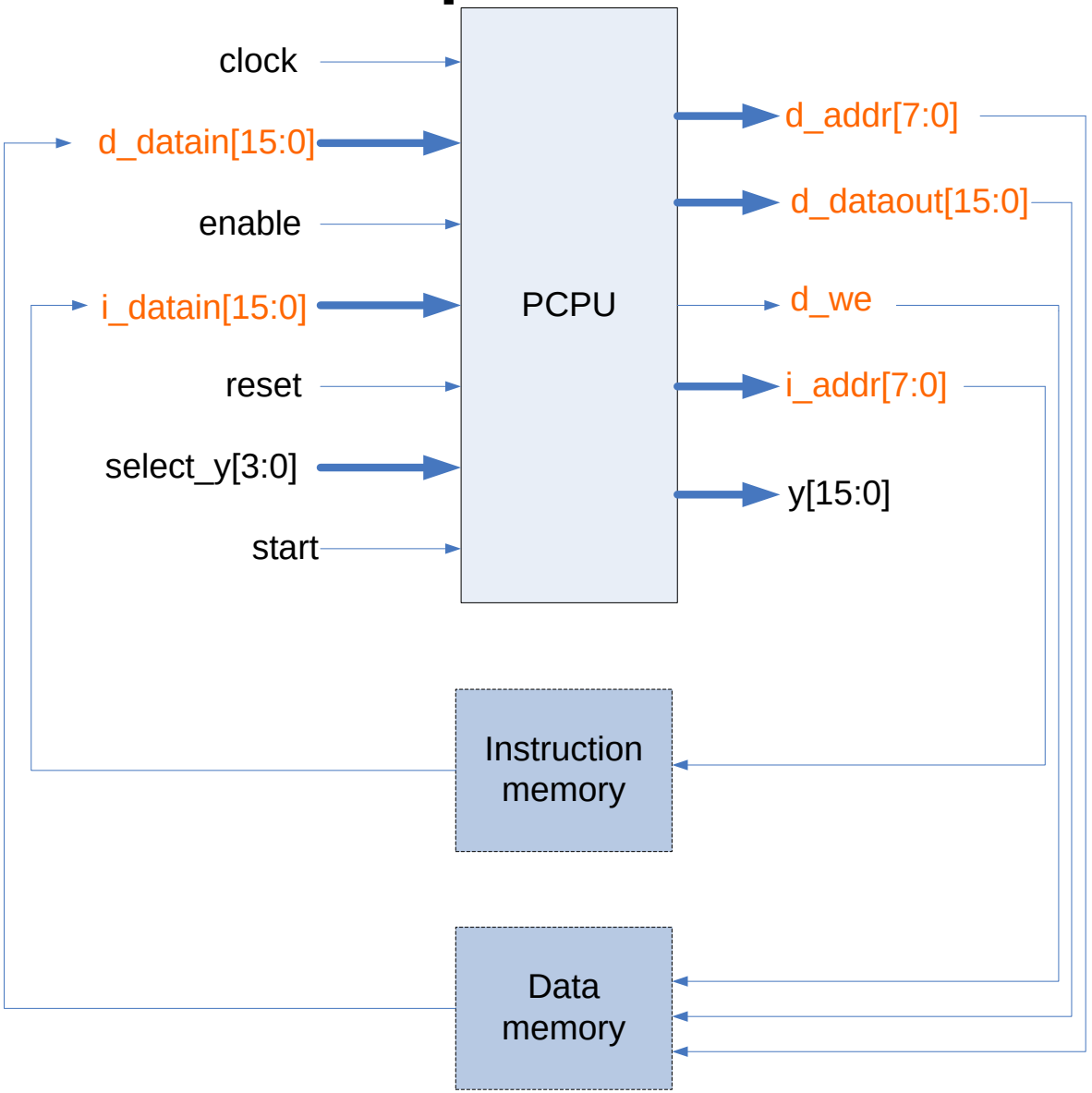
mnemonic	operand 1	operand2	operand3	op code	operation
JUMP		val2	val3	11000	jump to {val2, val3}
JMPR	r1	val2	val3	11001	jump to r1+{val2, val3}
BZ	r1	val2	val3	11010	if ZF=1 branch to r1+{val2, val3}
BNZ	r1	val2	val3	11011	if ZF=0 branch to r1+{val2, val3}
BN	r1	val2	val3	11100	if NF=1 branch to r1+{val2, val3}
BNN	…				
BC	r1	val2	val3	11110	if CF=1 branch to r1+{val2, val3}
BNC	…				

Block diagram

smdr: data for save memory directly



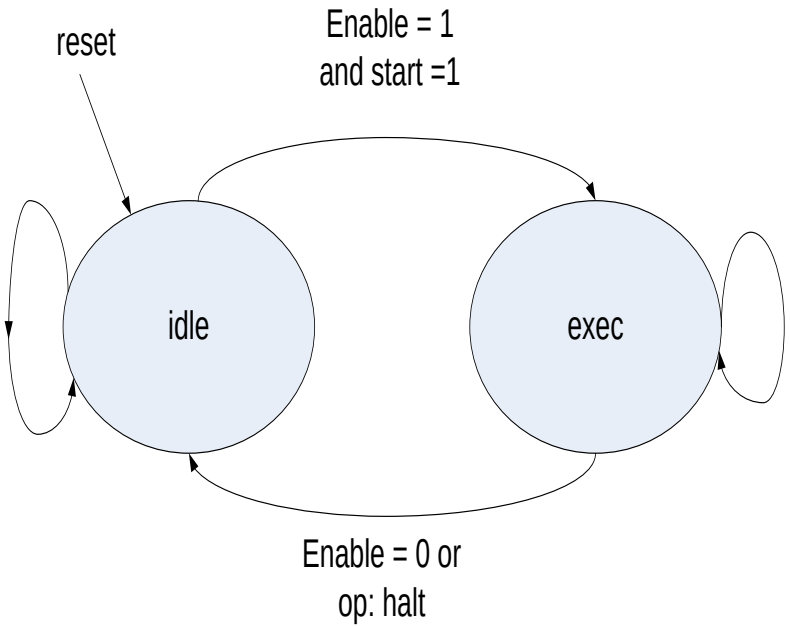
Top view


clock
d_datain[15:0]
enable
i_datain[15:0]
reset
select_y[3:0]
start
PCPU
d_addr[7:0]
d_dataout[15:0]
d_we
i_addr[7:0]
y[15:0]
Instruction memory
Data memory

Storage

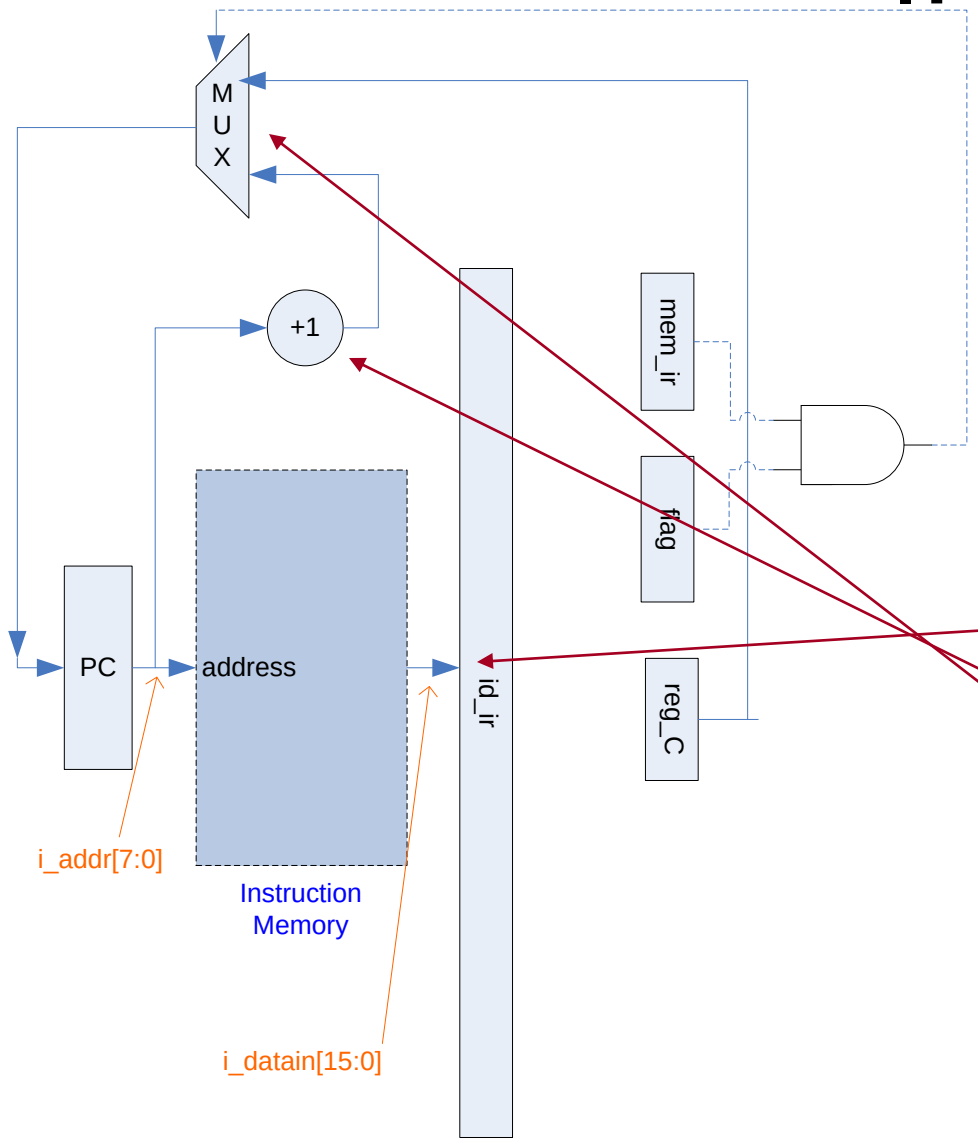
- Outside CPU core (SRAM, on-chip mem)
  - Instruction storage
    - Instruction memory (2^5 X 16b)
  - Data storage
    - Data memory (2^8 X 16b)
- Inside CPU core
  - Instruction storage
    - Stage instruction registers
      - id_ir(16b x1), ex_ir (16b x 1), mem_ir (16b x 1), wb_ir (16b x 1)
  - Data storage
    - General registers
      - Storage for operand data (2^3 X 16b)
    - Stage data registers
      - @ID: reg_A (16b x 1), reg_B (16b x 1), smdr (16b x 1),
      - @EX : reg_C (16b x 1), flag (1b x3), dw (1b x1), smdr1 (16b x 1),
      - @WB: reg_C1 (16b x 1)

CPU control




```
2   always @(posedge clock)
3      begin
4         if (!reset)
5            state <= `idle;
6         else
7            state <= next_state;
8      end
9
10  always @(*)
11     begin
12        case (state)
13           `idle :
14              if ((enable == 1'b1)
15              && (start == 1'b1))
16                 next_state <= `exec;
17              else
18                 next_state <= `idle;
19           `exec :
20              if ((enable == 1'b0)
21              || (wb_ir[15:11] == `HALT))
22                 next_state <= `idle;
23              else
24                 next_state <= `exec;
25        endcase
26     end
```

IF



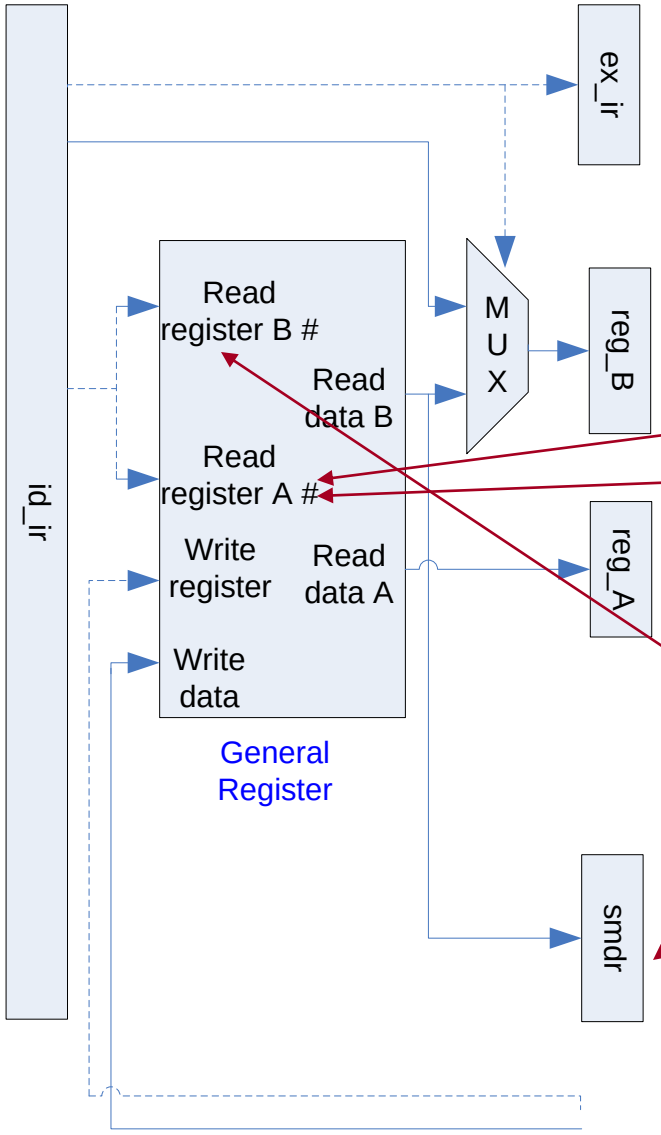

```
always @(posedge clock or negedge reset)
   begin
      if (!reset)
         begin
            id_ir <= 16'b0000_0000_\
            0000_0000;
            pc <= 8'0000_0000;
         end

      else if (state ==`exec)
         begin
            id_ir <= i_datain;

            if(((mem_ir[15:11] == `BZ)
               && (zf == 1'b1))
            || ((mem_ir[15:11] == `BN)
               && (nf == 1'b1)))
               pc <= reg_C[7:0];
            else
               pc <= pc + 1;
         end
```

ID

reg_A: 1st input operand of ALU, always gr[]

reg_B: 2nd input operand of ALU, gr[] or val

smdr: used only for STORE




```
else if (state == `exec)
   begin
      ex_ir <= id_ir;

      if ((id_ir[15:11] == `BZ)
      || (id_ir[15:11] == `BN))
         reg_A <= gr[(id_ir[10:8])];
      else
         reg_A <= gr[id_ir[6:4]]];

      if (id_ir[15:11] == `LOAD)
         reg_B <= {12'b0000_0000_0000, id_ir[3:0]};
      else if (id_ir[15:11] == `STORE)
         begin
            reg_B <= {12'b0000_0000_0000, id_ir[3:0]};
            smdr <= gr[id_ir[10:8]];
         end
      else if ((id_ir[15:11] == `BZ)
            || (id_ir[15:11] == `BN))
         reg_B <= {8'b0000_0000, id_ir[7:0]};
      else
         reg_B <= gr[id_ir[2:0]];
   end
```

operand1: r1

operand2: r2

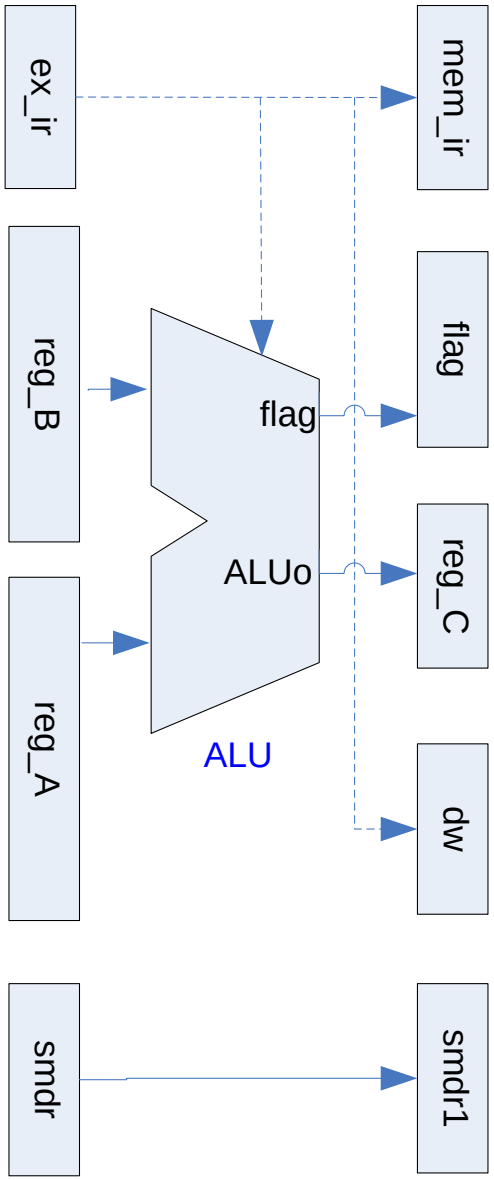
operand3: val3

Note latch!!

operand2,3: val2+val3

operand3: r3

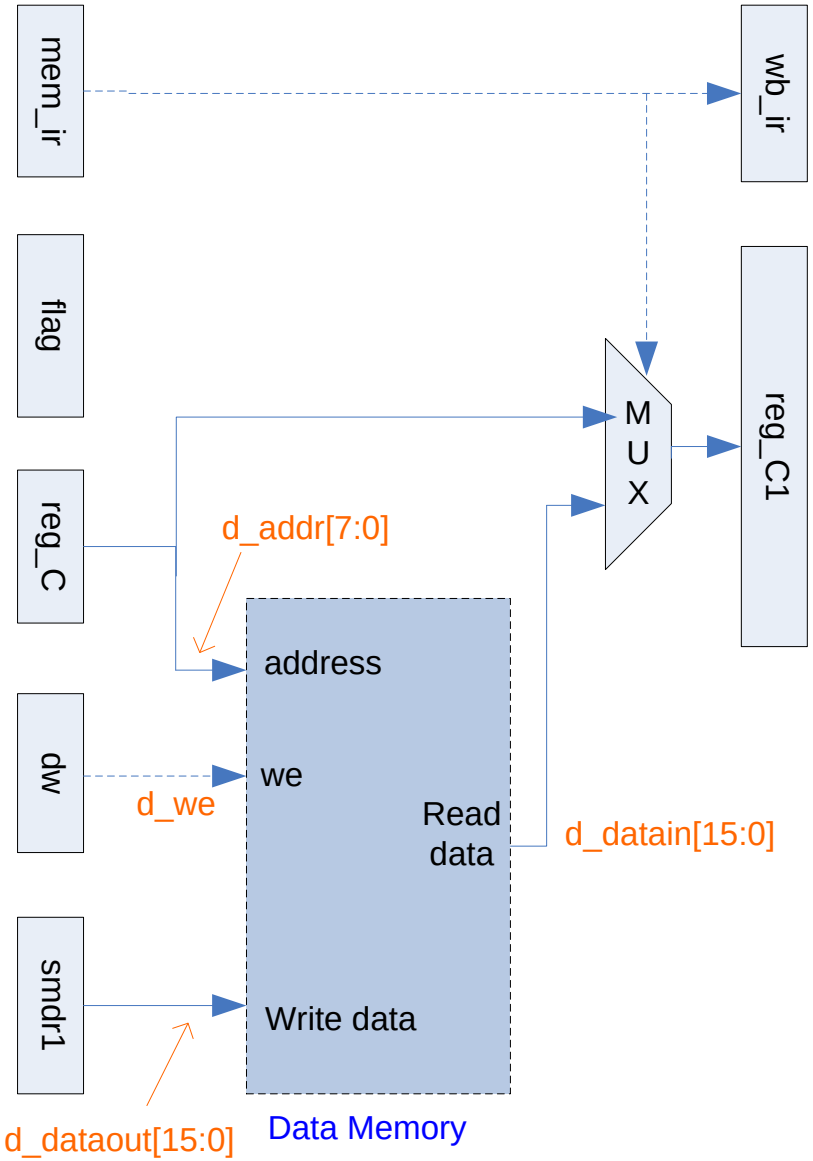
EX




```
else if (state == `exec)
   begin
      mem_ir <= ex_ir;
      reg_C <= ALUo;

      if ((ex_ir[15:11] == `ADD)
      || (ex_ir[15:11] == `CMP))
         begin
            if (ALUo == 16'b0000_0000_0000_0000)
               zf <= 1'b1;
            else
               zf <= 1'b0;
            begin
            if (ALUo[15] == 1'b1)
               nf <= 1'b1;
            else
               nf <= 1'b0;
         end
      if (ex_ir[15:11] == `STORE)
         begin
            dw <= 1'b1;
            smdr1 <= smdr;
         end
      else
         dw <= 1'b0;
   end
```

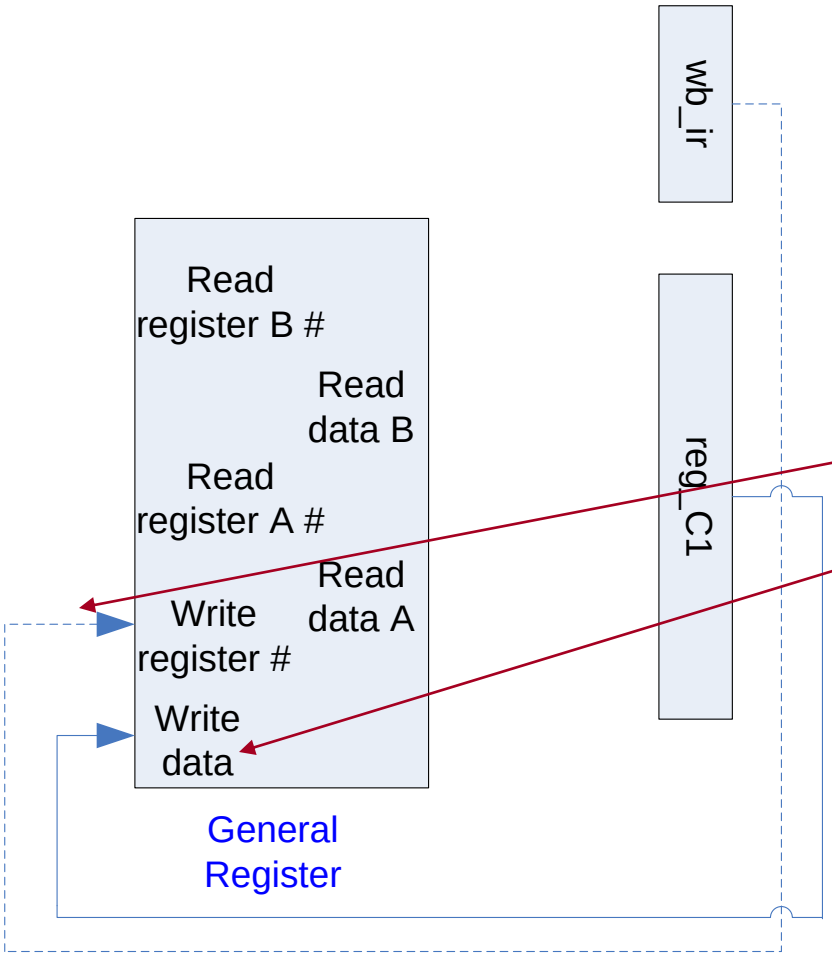
Note latch !

MEM




```
else if (state == `exec)
   begin
      wb_ir <= mem_ir;

      if (mem_ir[15:11] == `LOAD)
         reg_C1 <= d_datain;
      else
         reg_C1 <= reg_C;
   end
```

WB

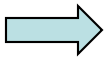



```
else if (state == `exec)
   begin
      if ((wb_ir[15:11] == `LOAD)
      || (wb_ir[15:11] == `ADD))
         gr[wb_ir[10:8]] <= reg_C1;
   end
```

Note latch !

Example of test pattern (texture simulation)

```
$display("pc:     id_ir      :reg_A:reg_B:reg_C:da:dd:  :w:reC1:gr1 :gr2 : gr3");
$monitor("%h:%b:%h:%h:%h:%h:%h:%b:%h:%h:%h:%h",
   pcpu.pc, pcpu.id_ir, pcpu.reg_A, pcpu.reg_B, pcpu.reg_C,
   d_addr, d_dataout, d_we, pcpu.reg_C1, pcpu.gr[1], pcpu.gr[2], pcpu.gr[3]);

enable <= 1; start <= 0; i_datain <= 0; d_datain <= 0; select_y <= 0;

#10 reset <= 0;
#10 reset <= 1;
#10 enable <= 1;
#10 start <=1;
#10 start <= 0;
   i_datain <= {`LOAD, `gr1, 1'b0, `gr0, 4'b0000};
#10 i_datain <= {`LOAD, `gr2, 1'b0, `gr0, 4'b0001};
#10 i_datain <= {`NOP, 11'b000_0000_0000};
#10 i_datain <= {`NOP, 11'b000_0000_0000};
   d_datain <=16'h00AB;  // 3 clk later from LOAD
#10 i_datain <= {`NOP, 11'b000_0000_0000};
   d_datain <=16'h3C00;  // 3 clk later from LOAD
#10 i_datain <= {`ADD, `gr3, 1'b0, `gr1, 1'b0, `gr2};
#10 i_datain <= {`NOP, 11'b000_0000_0000};
#10 i_datain <= {`NOP, 11'b000_0000_0000};
#10 i_datain <= {`NOP, 11'b000_0000_0000};
#10 i_datain <= {`STORE, `gr3, 1'b0, `gr0, 4'b0010};
#10 i_datain <= {`HALT, 11'b000_0000_0000};
```



```
LOAD gr1, gr0, 0
LOAD gr2,  gr0, 1
NOP
NOP
NOP
ADD gr3, gr1,  gr2
NOP
NOP
NOP
STORE gr3, gr0, 2
HALT
```

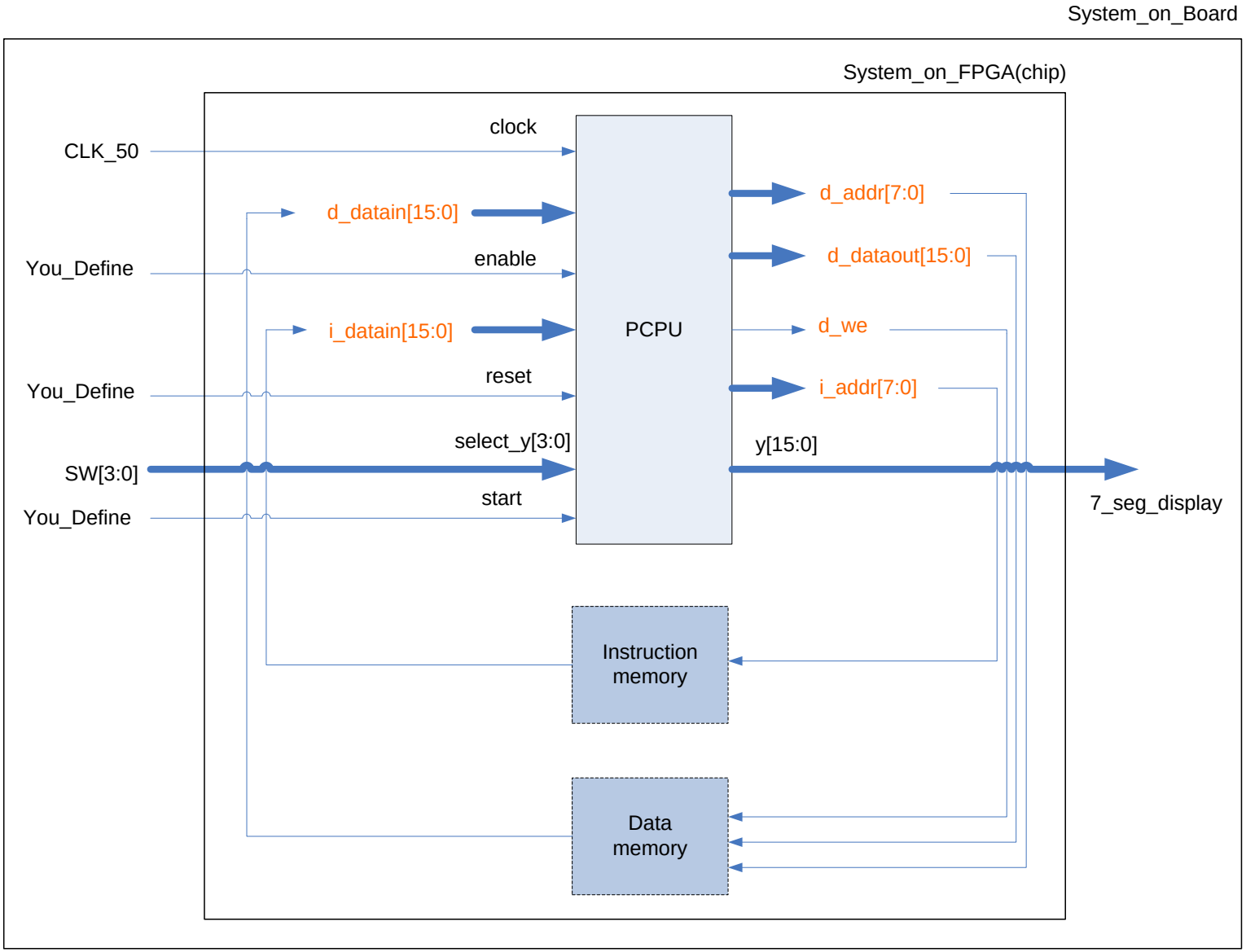
Simulation results (texture)

- Please test by yourself, and analyse it

```
LOAD gr1, gr0, 0
LOAD gr2,  gr0, 1
NOP
NOP
NOP
ADD gr3, gr1,  gr2
NOP
NOP
NOP
STORE gr3, gr0, 2
HALT
```

```
pc:    id_ir     :reg_A:reg_B:reg_C:da:dd:  :w:reC1:gr1 :gr2 : gr3
Finished circuit initialization process.
xx:xxxxxxxxxxxxxxx:xxxx:xxxx:xxxx:xx:xxxx:x:xxxx:xxxx:xxxx:xxxx
00:0000000000000000:0000:0000:0000:xx:xxxx:x:0000:0000:0000:0000
01:0001000100000000:0000:0000:xxxx:xx:xxxx:x:0000:0000:0000:0000
02:0001001000000001:0000:0000:xxxx:xx:xxxx:x:xxxx:0000:0000:0000
03:0000000000000000:0000:0001:0000:xx:xxxx:x:xxxx:0000:0000:0000
04:0000000000000000:0000:0000:0001:xx:xxxx:x:00ab:0000:0000:0000
05:0000000000000000:0000:0000:0001:xx:xxxx:x:3c00:00ab:0000:0000
06:0100001100010010:0000:0000:0001:xx:xxxx:x:0001:00ab:3c00:0000
07:0000000000000000:00ab:3c00:0001:xx:xxxx:x:0001:00ab:3c00:0000
08:0000000000000000:0000:0000:3cab:xx:xxxx:x:0001:00ab:3c00:0000
09:0000000000000000:0000:0000:3cab:xx:xxxx:x:3cab:00ab:3c00:0000
0a:0001101100000010:0000:0000:3cab:xx:xxxx:x:3cab:00ab:3c00:3cab
0b:0000100000000000:0000:0002:3cab:xx:xxxx:x:3cab:00ab:3c00:3cab
0c:0000100000000000:0000:0000:0002:xx:xxxx:x:3cab:00ab:3c00:3cab
0d:0000100000000000:0000:0000:0002:xx:xxxx:x:0002:00ab:3c00:3cab
0e:0000100000000000:0000:0000:0002:xx:xxxx:x:0002:00ab:3c00:3cab
0f:0000100000000000:0000:0000:0002:xx:xxxx:x:0002:00ab:3c00:3cab
```

Board evaluation

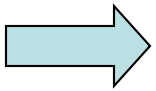


Memory implementation

- Write verilog by yourself
- Use **Core_Gernerate(ISE)/IP_Catalog(vivado)** to generate IP
  - Get the verilog built in
  - Memory initialization file (for testbench)
    - .mif (altera)
    - .coe (xilinx)
    - All available via course website
  - Study more on how to generate IP and use IP by website
    - eg., http://xilinx.eetop.cn/?action-viewnews-itemid-2516

Concept behind

- System on Chip
  - SOC
- From SOB to SOC
  - System on board to system on chip
- SOC
  - Processor
    - ARM/DSP/MIPS/X86
    - Program with C/C++
  - Memory
  - FPGA/ASIC
    - Your own verilog



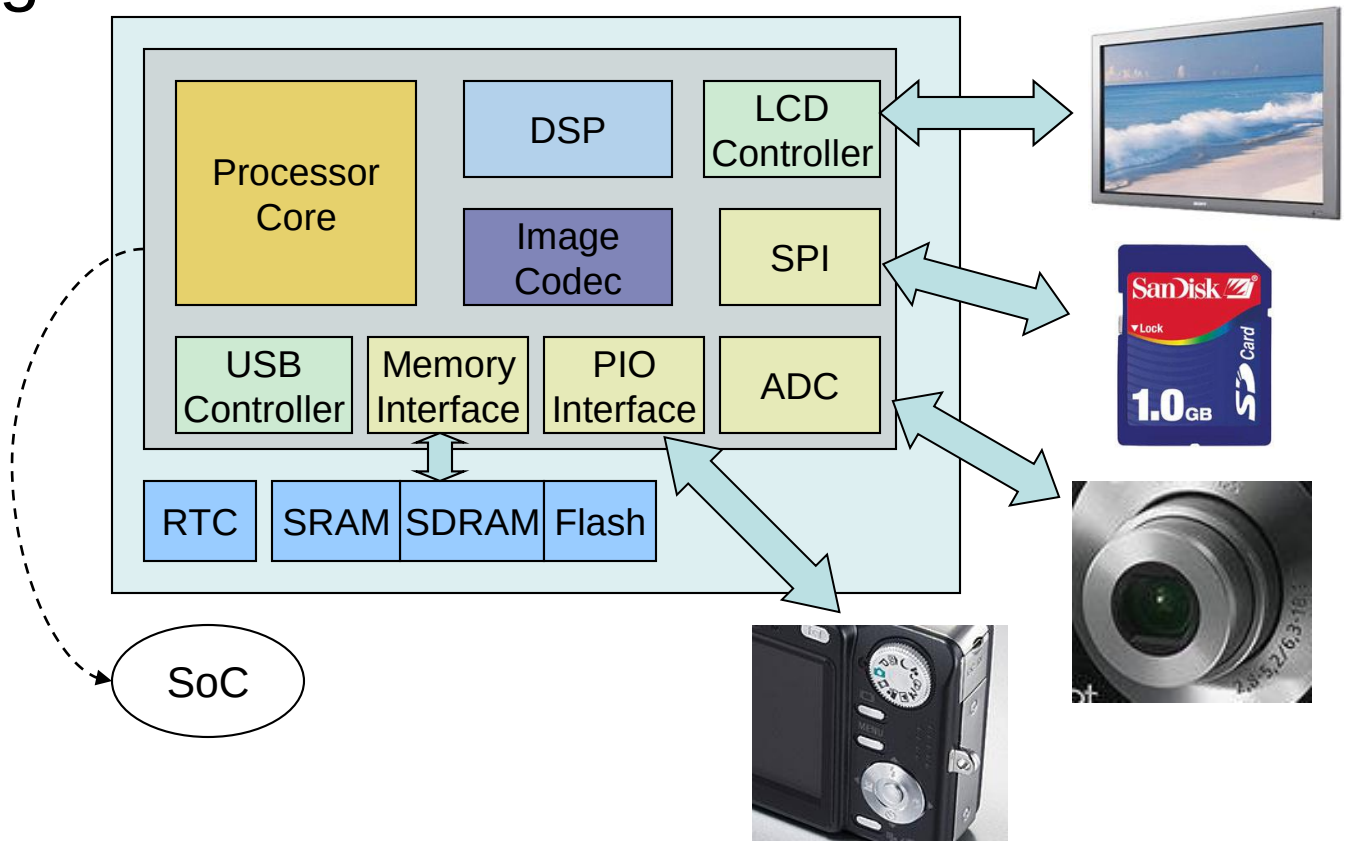
SOPC: (P: programmable)

Newly FPGA, the state-of-art all programmable device, can cover everything with low cost solution, and acceptable performance,

- Therefore, software and hardware division is a problem
  - New design methodology
    - SW/HW Co-design
    - Key of embedded system design

Embedded System Design

- Example: Digital camera hardware block diagram



Labs

- 1, Complete the designs above
  - Complete the verilog with variable declarations
  - Simulate by the test pattern (texture simulation), (not board verfication)
- 2, Add more to your verilog
  - Complete the left operations other than * ones
    - LDIH, ADDI,ADDC, …
  - Add flag register
    - CF
  - Add your original operations
    - You can add up to 5 ops with unused codes: 10011, 10100, 10101, 10110, 10111.
  - Design you own test patterns
- 3, Board verification
- Submit your report due to May 15th
  - Your own operation set description (like the table in slide#7/8/9), verilog, testbench
  - snapshot of simulation (texture only), RTL snapshots, Board running snapshot, design report (timing/area/power).
  - Impression (感想) on this project

Further projects
---Hazards

Hazards

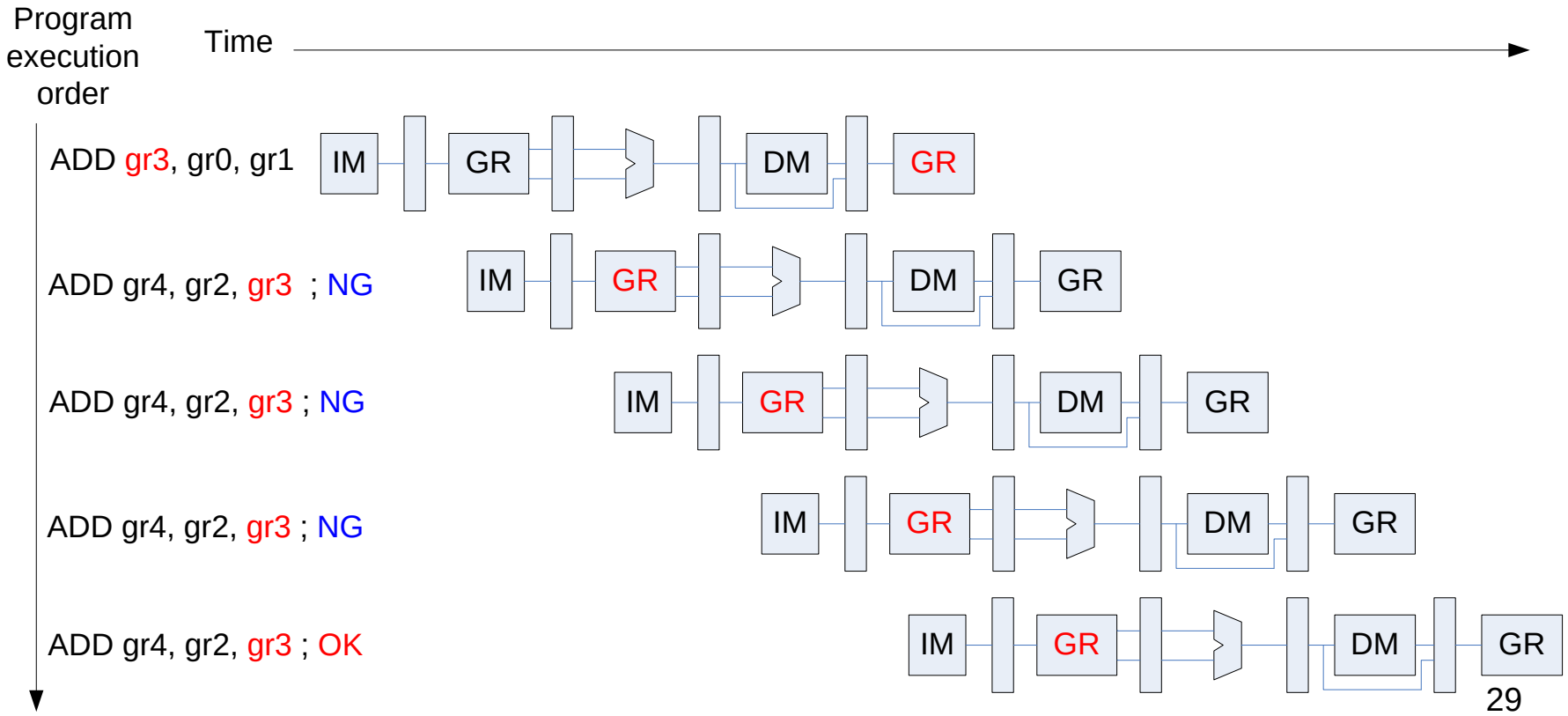
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

- Conflict for use of a resource
- In X86（by intel） pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
  - So called von Neumann structure
- In ARM/MIPS pipeline
  - separate instruction/data caches
  - So called Harvard structure

Data Hazards

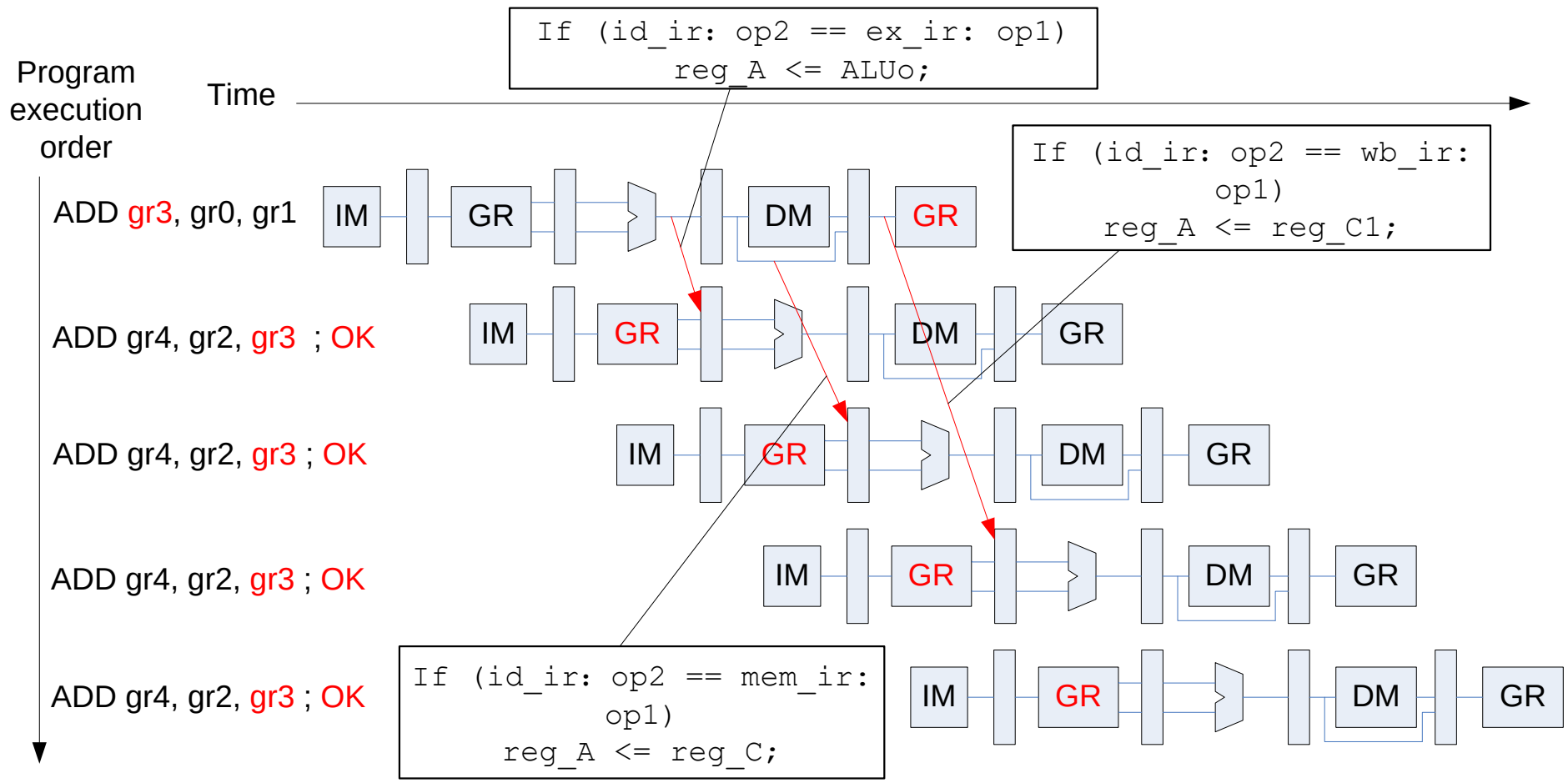
- An instruction depends on completion of data access by a previous instruction



Solution

- Software
  - Insert 3 NOPs
- Hardware
  - Data forward
    - For arithmetic op
  - Data forward & Stall
    - For LOAD

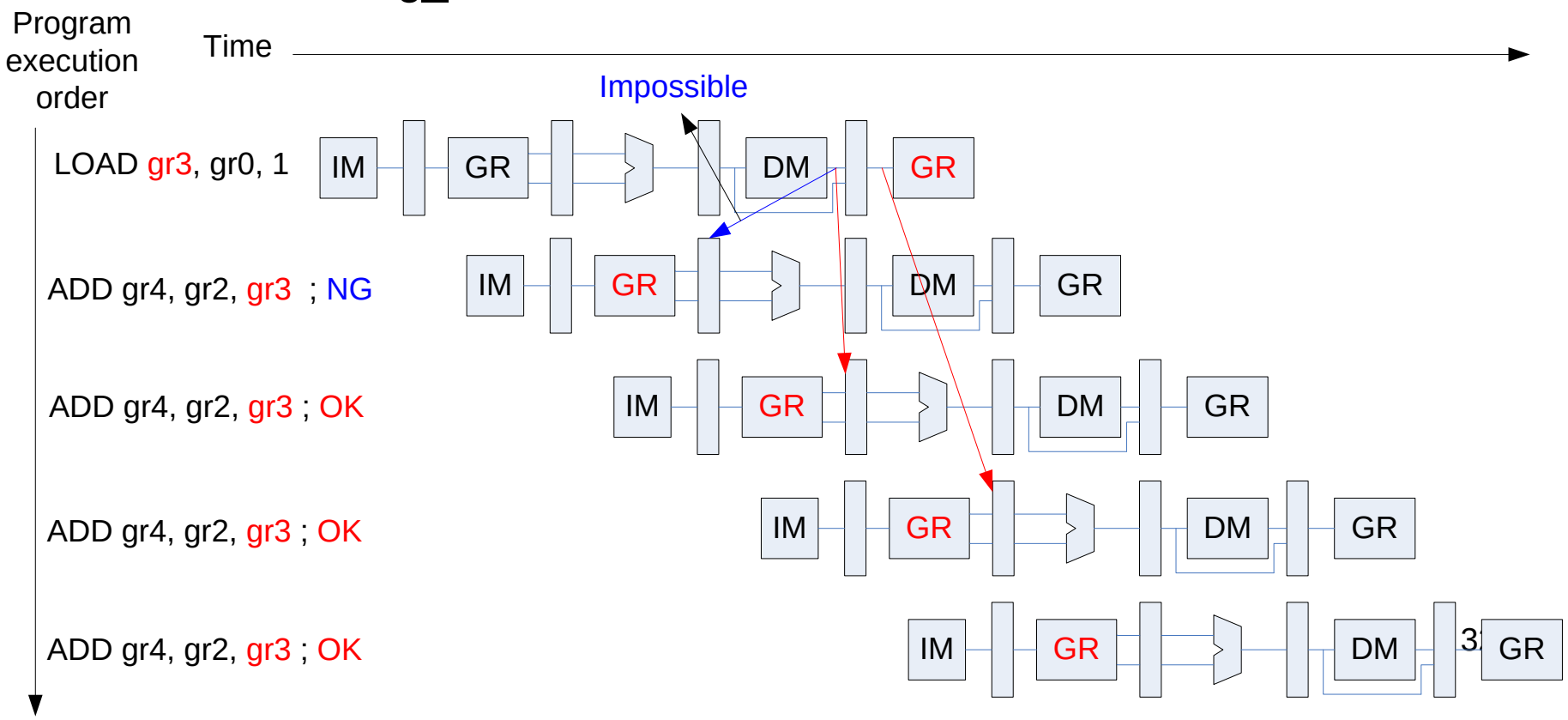
Data forwarding



Data forwarding happens in ID stage, always check *id_ir* to decide *reg_A/B*

Can’t always forwarding

- “LOAD” can still cause hazard
  - since data was accessed from data memory
    - after reg_C



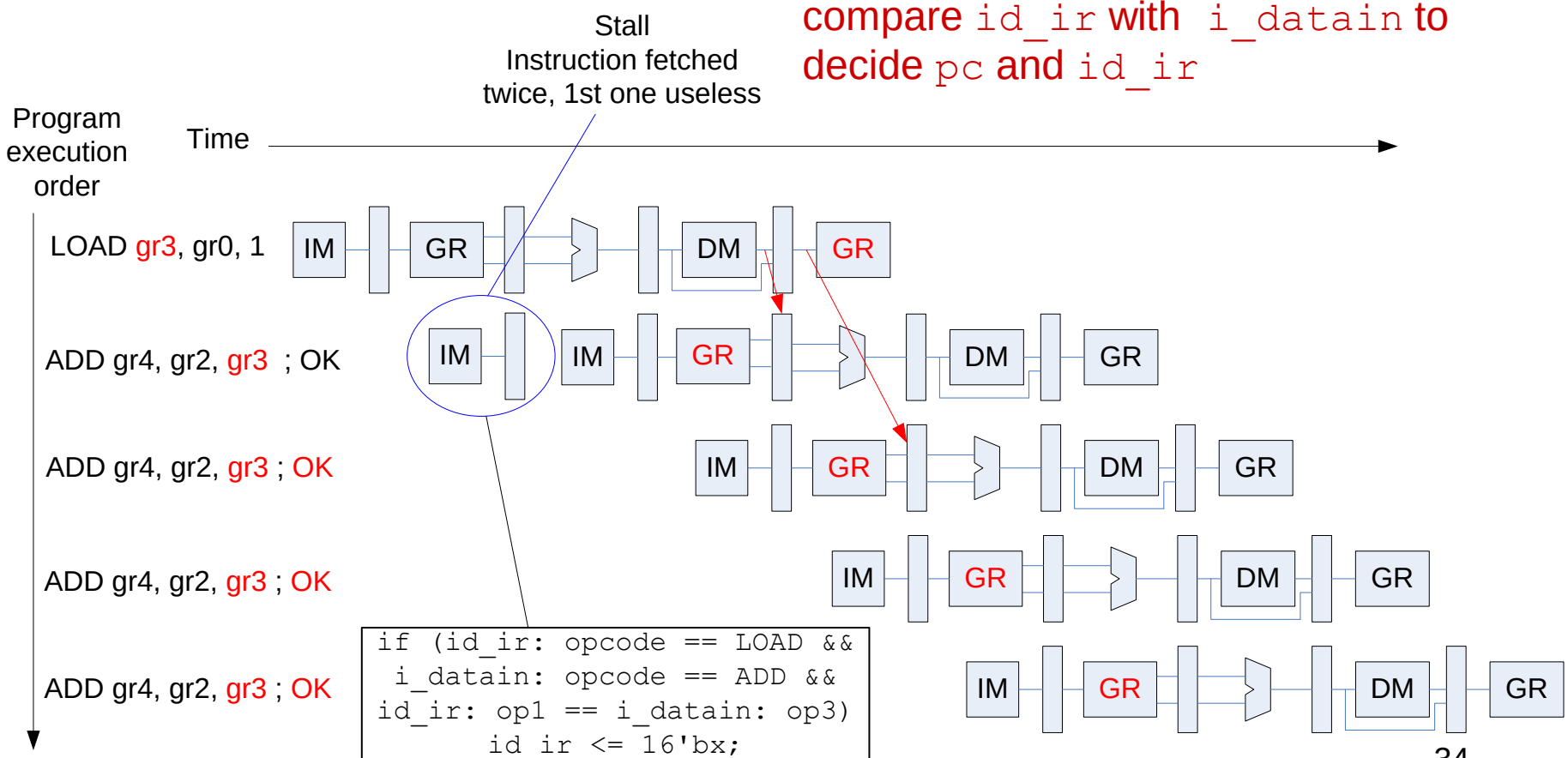
Solution

- Software
  - Insert 3 NOPs
- Hardware
  - Joint usage of Data forwarding & Stall
    - Data forwarding to solve 2nd and 3rd NOPs
    - Stall to sove 1st NOP

Stall

- Stall the pipeline by keeping an instruction in the same stage

Stall happens in IF stage, always compare `id_ir` with `i_datain` to decide `pc` and `id_ir`



Control Hazard

- When CPU decide to branch, other instructions are still in pipeline!!
  - During the time from branch instruction fetch to branch address generate (from IF to ID to EX to MEM, 3 stages),
  - the 3 instructions followed by branch are in pipeline.
    - The computing results by them are useless
    - Should be flushed
      - or will impact the following instructions computing

Solution

- Software
  - Insert 3 NOPs
  - Insert independent operations
- Hardware
  - Flushing…

Labs

- 1, Finish both the data and control hazard HW solution design by Verilog
- 2, pass the 复杂指令测试 show in the website.
- 3, Board verification（including 复杂指令测试）
- Submit your report due to Jun 29th
  - Your own verilog, testbench
  - snapshot of simulation (texture only), RTL snapshots, Board running snapshot, design report (timing/area/power).
  - Impression (感想) on this project

Summary

- Hazard
  - Structure
  - Data
    - Arithmetic
      - Software solution
        - » NOP
      - Hardware solution
        - » Data forwarding
    - LOAD
      - Software solution
        - » NOP
      - Hardware solution
        - » Stall+Data forwarding
  - Control
      - Software solution
        - » NOP
        - » Independent operation
      - Hardware solution
        - » Flushing