ELE-475 COMPUTER ARCHITECTURE

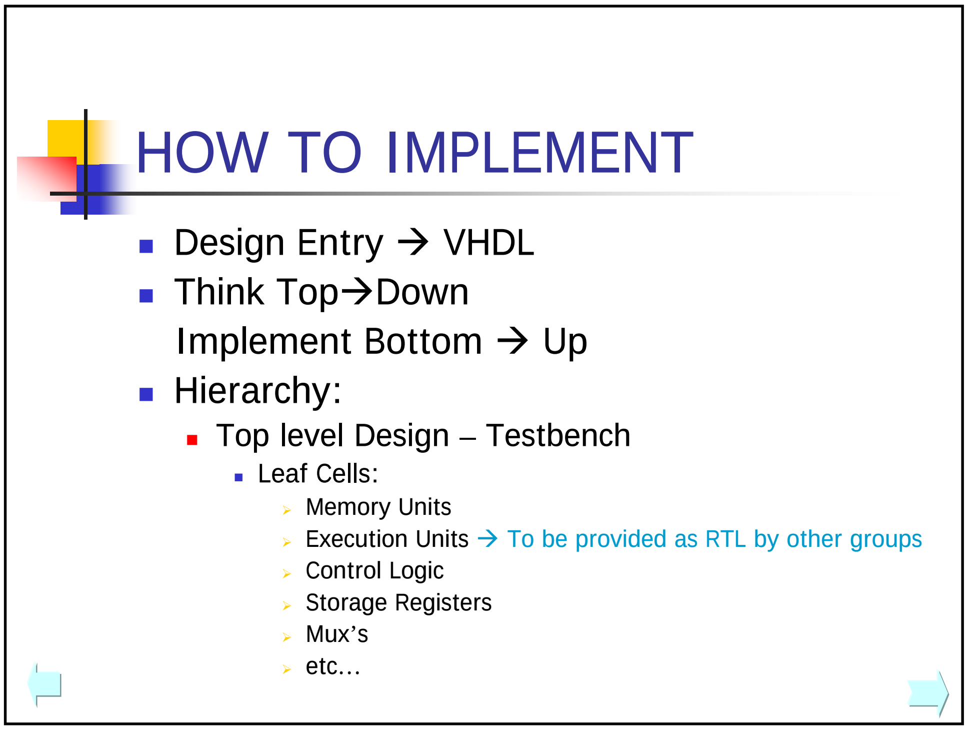
PLX 1.0 PROJECT – II

Zhenghong Wang & *Canturk Isci*

2-WAY SUPERSCALAR PROCESSOR in VHDL

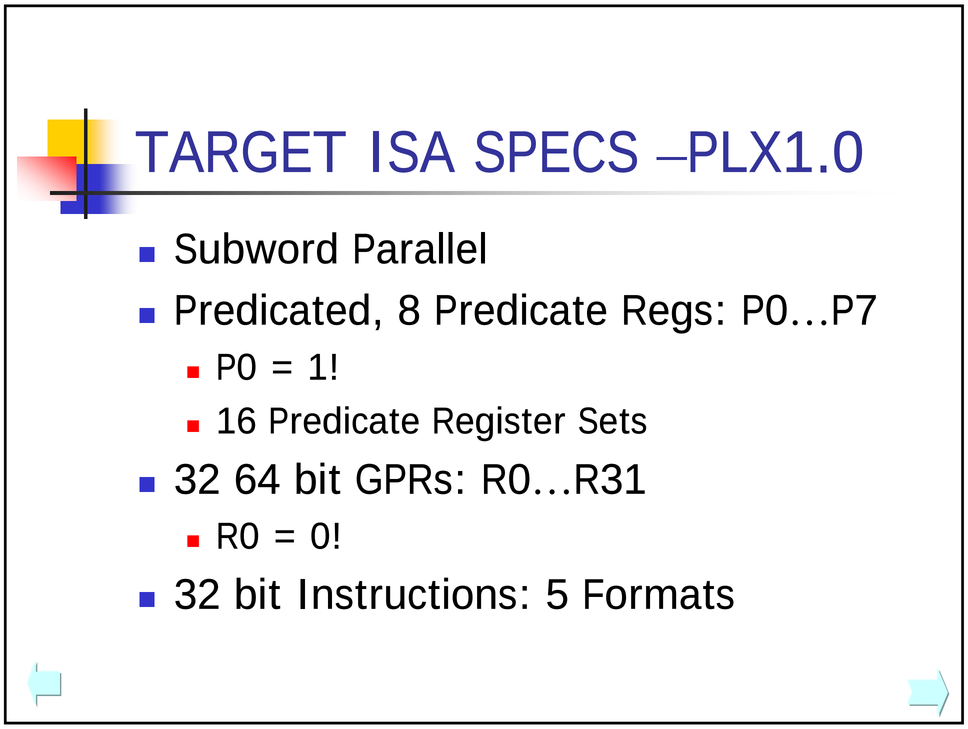
WHAT TO IMPLEMENT

- Pipelined 2-BANGER➔
  - ISA: PLX1.0
  - Hazard Detection
  - Bypassing Logic
  - Predication



HOW TO IMPLEMENT

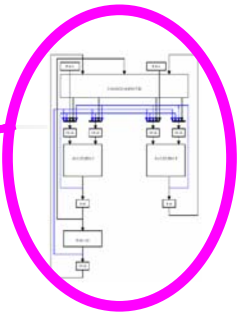
- Design Entry → VHDL
- Think Top→Down
  Implement Bottom → Up
- Hierarchy:
  - Top level Design – Testbench
    - Leaf Cells:
      - Memory Units
      - Execution Units → To be provided as RTL by other groups
      - Control Logic
      - Storage Registers
      - Mux's
      - etc…



TARGET ISA SPECS –PLX1.0

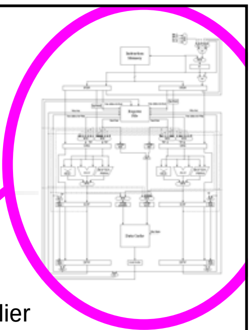
- Subword Parallel
- Predicated, 8 Predicate Regs: P0…P7
  - P0 = 1!
  - 16 Predicate Register Sets
- 32 64 bit GPRs: R0…R31
  - R0 = 0!
- 32 bit Instructions: 5 Formats

DESIGN OVERVIEW

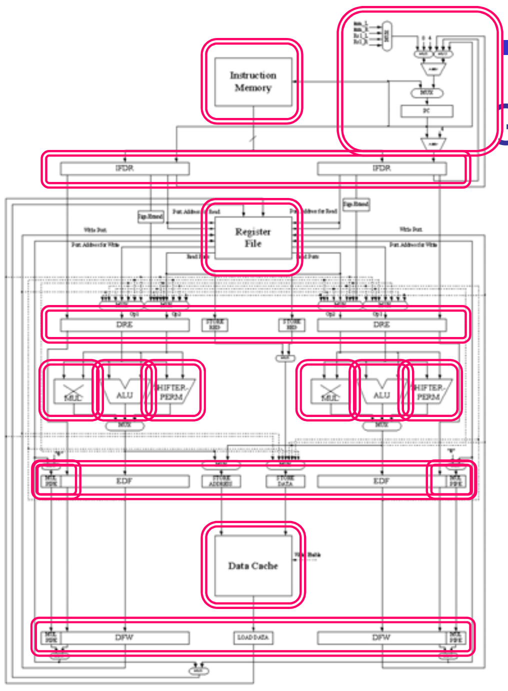


- Generic 2-Way Superscalar?
  - 2 Execution Pipes ✓
    - ALU/Shift-Permute/Multiplier
  - Single Load/Store Pipe ✓
  - 7 Port RegFile ✓
  - Out of order Completion ×
    - WAW hazards may still occur
    - 3 Register write ports still needed

DESIGN OVERVIEW



- ACTUAL DESIGN:
  - 2 Symmetrical Execution Pipes
    - Each with 1 ALU, 1 Shift-Permute, 1 Multiplier
      - Multiplier Takes 3 Cycles – Pipelined -, ALU and Shift-Permute Single Cycle
  - Single Load Store Pipe ⇔ Single Data Cache
  - Single 7 Port Register File
  - Single Instruction Memory
  - Single Predicate Register File
  - Memory Reference can be from each pipe
  - Standard Bypassing
    - One Special W→E bypass [Later]
    - Forwarding for Predicate Registers [Later]
  - Control Signals are Pipelined as well
    - Additional Pipe Registers for Multiplier Control [Later]

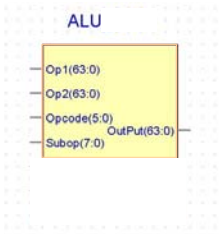


GN

- TOP LEVEL DESIGN
  - Instruction Memory
  - PC Generator
  - IF/DR Pipe Regs
  - GPR File
  - DR/E Pipe Regs
  - Multipliers
  - ALU Units
  - Shift-Permute Units
  - E/DF Pipe Regs
  - Data Cache
  - DF/W Pipe Regs
  - Additional Pipe Regs for Multipler Control
  - W→DF Bypass
  - W→E Bypass
  - DF→DF Bypass
  - DF→E Bypass
  - E→E Bypass

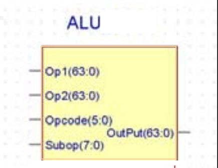
<LEAF LEVEL> COMPONENTS

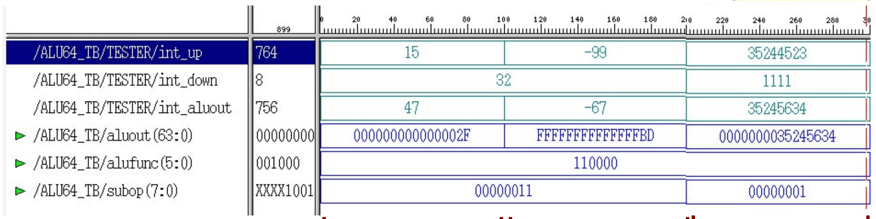
- 1) ALU
  - Behavioral Level
  - Performs required functions for the testbenches
    - padd.sw
    - cmp
    - psub.sw
    - loadi.hi/lo
    - load.8.update
    - load.8
    - store.8.update



COMPONENTS

- 1) ALU - Simulation

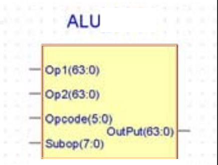


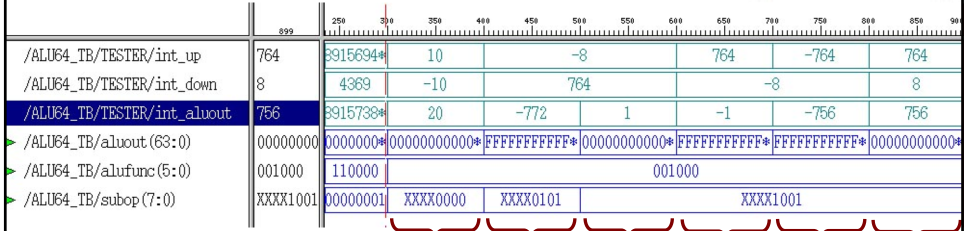

```
                              899  0  20  40  60  80  100  120  140  160  180  200  220  240  260  280  300
/ALU64_TB/TESTER/int_up       764       | 15               | -99              | 35244523
/ALU64_TB/TESTER/int_down     8         | 32                                  | 1111
/ALU64_TB/TESTER/int_aluout   756       | 47               | -67              | 35245634
/ALU64_TB/aluout(63:0)        00000000  | 000000000000002F | FFFFFFFFFFFFFFBD | 0000000035245634
/ALU64_TB/alufunc(5:0)        001000    | 110000
/ALU64_TB/subop(7:0)          XXXX1001  | 00000011                            | 00000001
```

$\underbrace{\quad}_{\text{padd.8 15, 32}}$ $\underbrace{\quad}_{\text{padd.8 -99, 32}}$ $\underbrace{\quad}_{\text{padd.2 x35\_24\_45\_23, x00\_00\_11\_11}}$

COMPONENTS

- 1) ALU - …Simulation

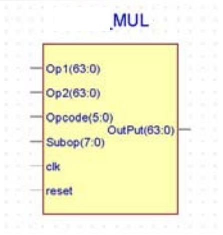



```
                              899  250  300  350  400  450  500  550  600  650  700  750  800  850  900
/ALU64_TB/TESTER/int_up       764       | 8915694* | 10           | -8           |              | 764          | -764         | 764
/ALU64_TB/TESTER/int_down     8         | 4369     | -10          | 764                         | -8                          | 8
/ALU64_TB/TESTER/int_aluout   756       | 8915738* | 20           | -772         | 1            | -1           | -756         | 756
/ALU64_TB/aluout(63:0)        00000000  | 0000000* | 00000000000* | FFFFFFFFFFF* | 00000000000* | FFFFFFFFFFF* | FFFFFFFFFFF* | 00000000000*
/ALU64_TB/alufunc(5:0)        001000    | 110000   | 001000
/ALU64_TB/subop(7:0)          XXXX1001  | 00000001 | XXXX0000     | XXXX0101     | XXXX1001
```

$\underbrace{\quad}_{\text{cmp.eq 10,-10}}$ $\underbrace{\quad}_{\text{cmp.ge -8,764}}$ $\underbrace{\quad}_{\text{cmp.geu -8,764}}$ $\underbrace{\quad}_{\text{cmp.geu 764,-8}}$ $\underbrace{\quad}_{\text{cmp.geu -764,-8}}$ $\underbrace{\quad}_{\text{cmp.geu 764,8}}$

COMPONENTS

- 2) MULTIPLIER
  - Behavioral Level
  - Performs:
    - pmul.odd
    - pmul.odd
  - Requires Reset & Clk for the 3 stage pipe

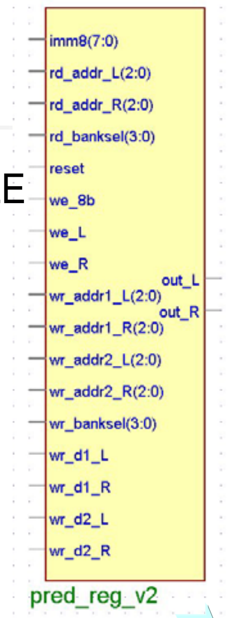


COMPONENTS

- 3) SHIFT-PERMUTE UNIT
  - Behavioral Level
  - Provided by Group1

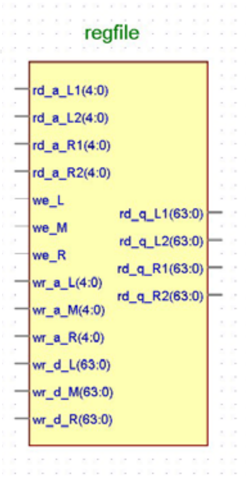
COMPONENTS

- 4) PREDICATE REGISTER FILE
  - 16 predicate banks,
    - 4 bit address to specify
    - each bank → 8 bit predicate set
  - 2 Read Ports, for Pi,Pj
    - 2 3-bit read addresses
  - 4 Bitwise writes for Pi,Pj and Pk,Pl
    - 4 3-bit write addresses
  - 1 Byte write port
    - 1 4-bit write address



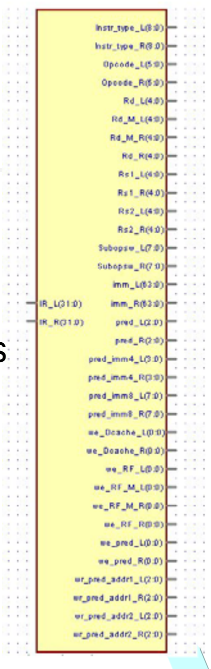
COMPONENTS

- 5) GPR FILE
  - 32 64-bit Registers
  - 4 Read Ports
    - 4 5-bit addresses
  - 3 Write Ports
    - 3 5-bit addresses



COMPONENTS

- 6) INSTRUCTION DECODER
  - Interprets/Decodes Instructions
  - Separates the Instruction Filelds
    - Opcode, Subop, Rd, Rs, Imm, etc.
  - Sets/clears write-enable bits for different instructions



COMPONENTS

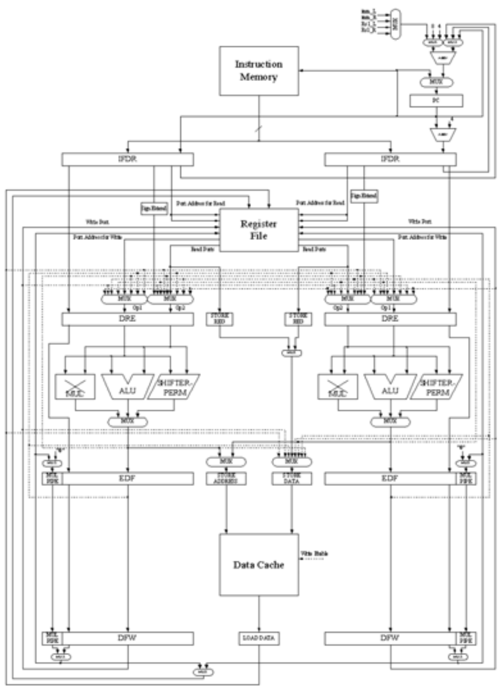
- 7) PIPELINE REGISTERS
  - Pipe Data through Datapath
  - Pipe Control Signals through Control Path
- 8) CONTROLLER
  - Pure Combinational Logic
  - Checks the piped instruction fields and predicates to detect hazards/stalls/bypasses

COMPONENTS

- 9) INSTRUCTION MEMORY
  - 32-words, Big-Endian, Byte Addressed
    - 1Kb → 256 wordlines
  - Aligned Addressing (!jmp imm → multiple of 4)
  - Initialized from “Instruction_image.ini”
- 10) DATA MEMORY
  - Similar to Instruction Memory
  - Single write/read address

Pipeline Organization

- Two symmetrical pipelines
- Standard data forwarding logic for general purpose register file
- Data forwarding logic for predicate register file
- Special pipeline register for PMUL instructions



Symmetrical pipelines

- Two symmetrical pipelines:
  - Each includes 1 ALU, 1 SHF and 1 MUL
- They share 1 data cache and 1 LD/ST pipe, thus need data merging unit
- Data merging is done in E stage

E-stage data merging

- Reasons for E-stage data merging-I
  - Simpler Control Logic: though the decision can be made in DR stage, it needs complex condition-match checking logic. This is mainly because the validity of an instruction can not be completely determined until in E stage.

E-stage data merging

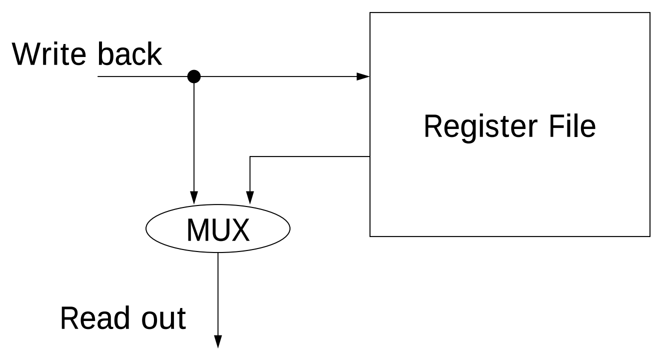
- Reasons for E-stage data merging-2
  - To reduce the length of critical path in DF stage:data cache is the slowest component in the processor. Put data merging logic in DF stage will lengthen the critical path in DF stage and may result in longer cycle time. Indeed, in our design, no other component is connected with cache in serial.

Data Forwarding: GPRF

- Standard data forwarding path:
  E-E, E-DF, DF-E, DF-DF forwarding
- Special data forwarding path:
  W-E forwarding

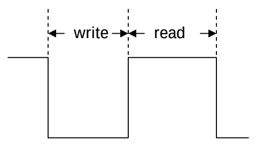
W-E forwarding

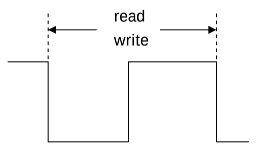
- Equivalent Structure



W-E forwarding

- Advantages:
  - Avoid the 2-phase write/read operation cycle: overlapped read and write operations
  - Considerably shorten the length of critical path in DR stage, and comparators in DR may be possible???

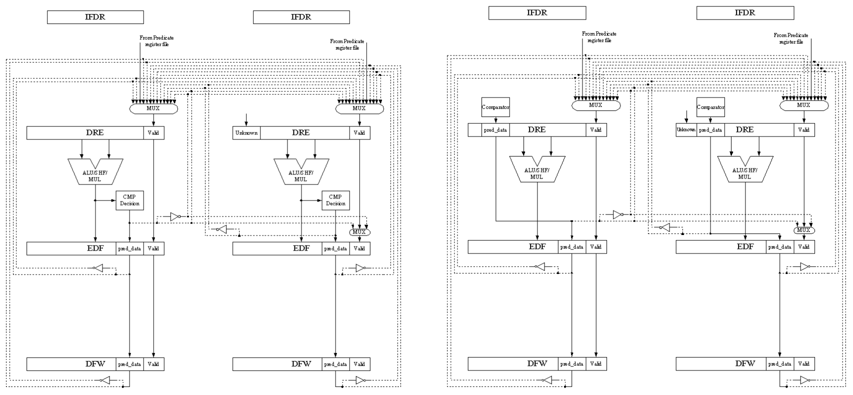




A bug in data forwarding

We found a bug for JMP.reg instruction.
We didn’t implement data forwarding path to JMP address calculator.
We will fix it after the presentation.

Data forwarding: Pred Reg



Data Forwarding: Pred Reg

- Where to place the comparators?
  - Use ALU as “comparator”: Compare in E stage

    pros: less hardware, operations are regular and no “side-effect”;

    cons: maybe the long critical path for data forwarding;

  Possible solution: use faster comparator in parallel with ALU, i.e., in E stage, to reduce the length of the critical path

Data Forwarding: Pred Reg

- Where to place the comparators?
  - Use special comparators in DR stage:

    pros: short critical path for data forwarding;

    cons: long critical path in DR stage: its operands should come from the outputs of the MUXs to receive correct source data(forwarded data)

Data Forwarding: Pred Reg

- Types of forwarding path:
  - E-E forwarding
  - E-DF forwarding
  - DF-E forwarding
  - W-E forwarding
- E-DF forwarding and “unknown” bit

E-DF forwarding & ”unknown”

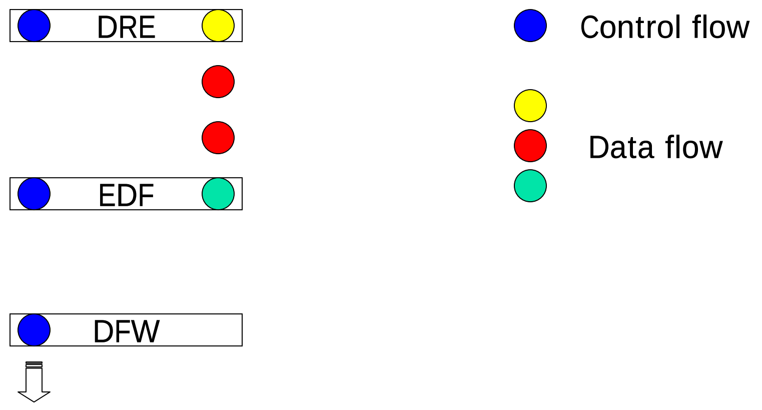
Left way | Right way

P0: CMP R1,R2,P1,P2 | P1: some instruction

If the above condition is satisfied, one can not determine whether the instruction in right way is valid or not until in E-stage. So first in DR stage, we set the unknown bit, then in E stage, if “unknown” is true, the E-DF forwarding is selected.

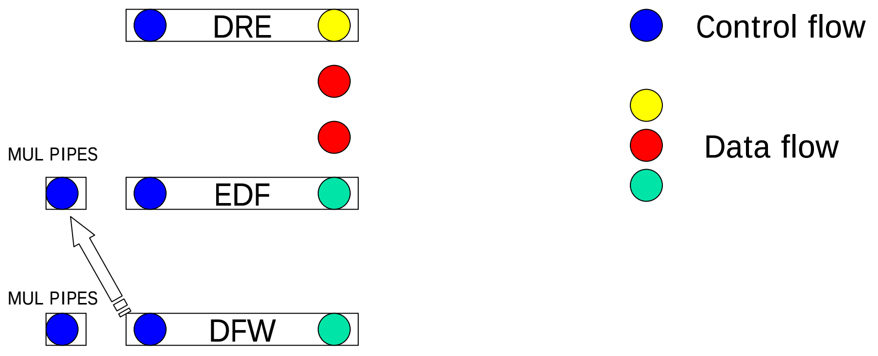
MUL PIPES

- Why use MUL PIPEs?



MUL PIPES

- How do MUL PIPEs work?



Hazard Detection

- Structural Hazards requiring stalls
  - Hazards in LD/ST pipe
  - Hazards related to PMUL instructions

IF	DR	E	E	E	DF	W
	IF	DR	E	DF	W	
		IF	DR	E	DF	W

Hazard Detection

- Data Hazards requiring stalls
  - Instructions that cause pipeline interlock

    1) LD   2) PMUL

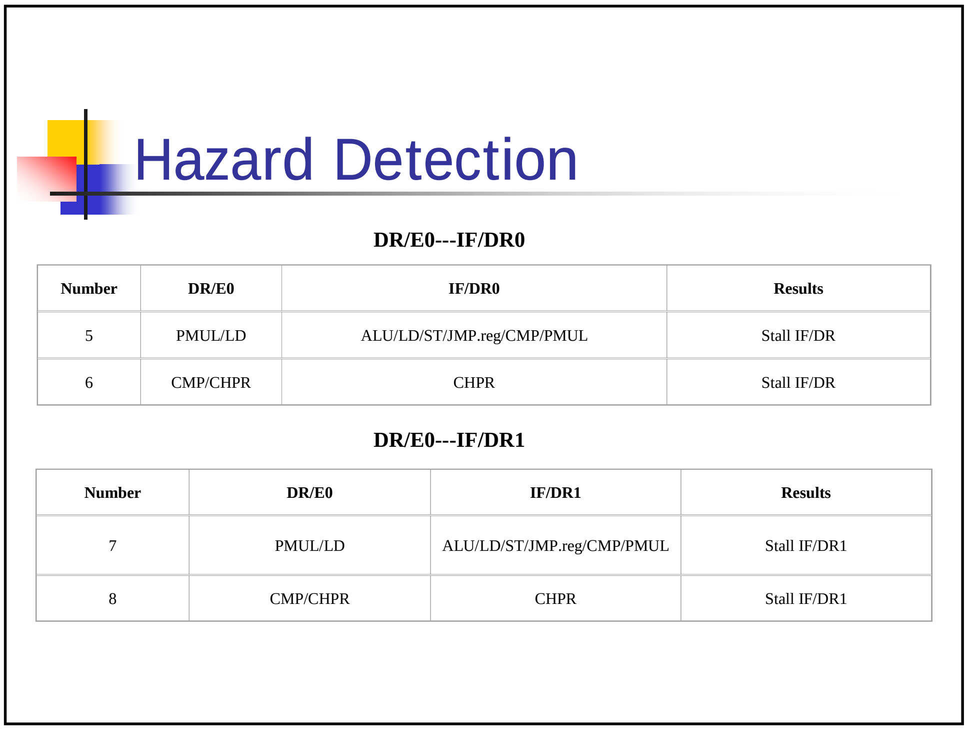
    3) CMP followed by JMP/Changepr
- Control Hazards

  No stalls are caused by JMPs. Actually, they work in a predict-untaken manner in our implementation

Hazard Detection

IF/DR0---IF/DR1

Number	IF/DR0	IF/DR1	Results
1	ALU/PMUL/LDi/LD/ST.upd	ALU/LD/ST/JMP.reg/CMP/PMUL	Stall IF/DR1
2	LD/ST	LD/ST	Stall IF/DR1
3	CMP/CHPR	JMP/CHPR	Stall IF/DR1
4	CHPR	any	Stall IF/DR1



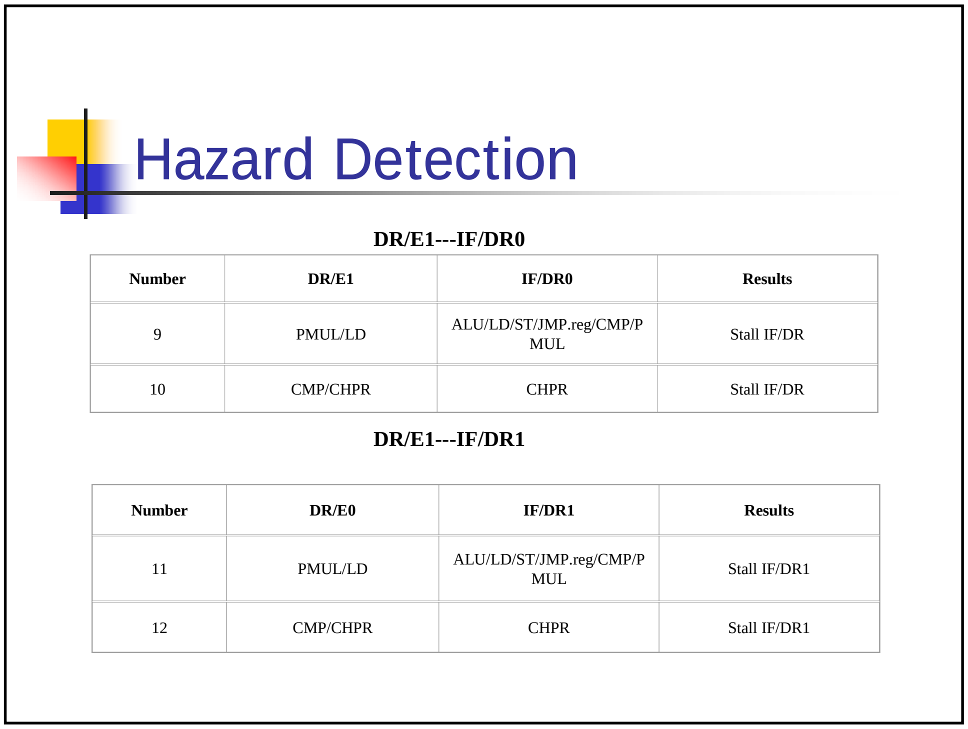
Hazard Detection

DR/E0---IF/DR0

Number	DR/E0	IF/DR0	Results
5	PMUL/LD	ALU/LD/ST/JMP.reg/CMP/PMUL	Stall IF/DR
6	CMP/CHPR	CHPR	Stall IF/DR

DR/E0---IF/DR1

Number	DR/E0	IF/DR1	Results
7	PMUL/LD	ALU/LD/ST/JMP.reg/CMP/PMUL	Stall IF/DR1
8	CMP/CHPR	CHPR	Stall IF/DR1



Hazard Detection

DR/E1---IF/DR0

Number	DR/E1	IF/DR0	Results
9	PMUL/LD	ALU/LD/ST/JMP.reg/CMP/P MUL	Stall IF/DR
10	CMP/CHPR	CHPR	Stall IF/DR

DR/E1---IF/DR1

Number	DR/E0	IF/DR1	Results
11	PMUL/LD	ALU/LD/ST/JMP.reg/CMP/P MUL	Stall IF/DR1
12	CMP/CHPR	CHPR	Stall IF/DR1

Hazard Detection

E/DF0---IF/DR0

Number	E/DF0	IF/DR0	Results
13	PMUL	ALU/LD/ST/JMP.reg/CMP/PMUL	Stall IF/DR
14	PMUL	any instruction that will write RF except PMUL insructions	Stall IF/DR

E/DF0---IF/DR1

Number	E/DF0	IF/DR1	Results
15	PMUL	ALU/LD/ST/JMP.reg/CMP/PMUL	Stall IF/DR1

Hazard Detection

E/DF1---IF/DR0

Number	E/DF1	IF/DR0	Results
16	PMUL	ALU/LD/ST/JMP.reg/CMP/PMUL	Stall IF/DR

E/DF1---IF/DR1

Number	E/DF1	IF/DR1	Results
17	PMUL	ALU/LD/ST/JMP.reg/CMP/PMUL	Stall IF/DR
18	PMUL	any instruction that will write RF except PMUL insructions	Stall IF/DR

Hazard Detection: For JMPs

IF/DR0---IF/DR1

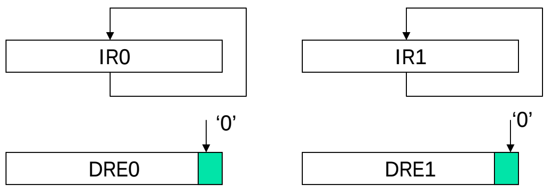
Number	IF/DR0	IF/DR1	Results
1	JMP	any	Cancel IF/DR1(only if JMP is valid and taken)

DR/E0 or 1---IF/DR0 or 1

Number	DR/E0 or 1	IF/DR0 or 1	Results
2	JMP	any	Cancel IF/DR(only if JMP is valid and taken)

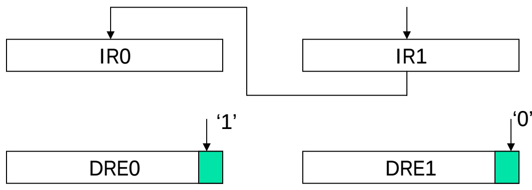
Stall and Cancel Operations

- What will happen when stall?
  - Stall IFDR0:
    - Nullify current instructions in IR0 and IR1
    - PC <= PC

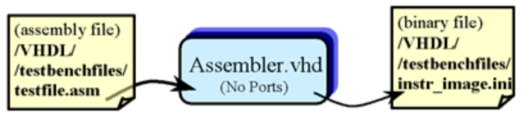


Stall and Cancel Operations

- What will happen when stall?
  - Stall IFDR1:
    - Nullify instruction in IR1
    - $IR0_{new}$ <= $IR1_{old}$, $IR1_{new}$ <= next instruction
    - PC <= PC+4



THE ASSEMBLER



- Generates the Binary Instruction Sequence for the Instruction Memory
- Easily Integrated with Testbench
- Flexible Assembly File Format >>
- Informative Error Diagnostics >>

THE ASSEMBLER

- File Format:

```
#OUR ASSEMBLY FILE:

P7:cmp.leu R1, R11, P1,P0
  P6 :     loadi.hi R6, -9
p0: PADD.8.u r5 , r3, r2   #comment
P2:jmp.link -1
```

- ➔Can have comment lines
- ➔Can have blank lines
- ➔Can have spaces between operands
- ➔Can have indentation and spaces between predicate fields
- ➔Case insensitive and can have comments after instructions

THE ASSEMBLER

- Error Diagnostics:

Erroneous Instruction:	Generated Error Message by Assembler:
PADD.8.u r5 , r3, r2	***ERROR***: in line -> 6 "Invalid instruction[non numeric Predicate id] -->PADD.8.u r5 , r3, r2
P3:padd.5.u r5, r4, r3	***ERROR***: in line -> 7 "Invalid instruction [wrong subword size field for ...] -->padd.5.u r5, r4, r3"
P8:psub.4.s r5, r4, r3	***ERROR***: in line -> 7 "Invalid instruction[non numeric Predicate id] -->P8:psub.4.s r5, r4, r3"
P4:psub.4. r5, r4, r3	***ERROR***: in line -> 11 "Invalid instruction [expected u or s for psub] -->P4:psub.4. r5, r4, r3"
P6 : loadi.hi 6, -9	***ERROR***: in line -> 5 "Invalid instruction[expected Rd register field] -->P6 : loadi.hi 6, -9"

THE ASSEMBLER

- See web page for simulator output and generated binary file
- Yet some instructions are not implemented
- Bugs – Deficiencies:
  - Cannot Recognize tab separators
  - Immediate range: -99999 – 99999
  - P0:jmp.reg R18, 0

    → To fit Type-1 instruction Format

FINAL TESTBENCH

- Initialize the Instruction Memory
  - Assembler → Binary File
  - Read Binary File
    - Or Directly Assembler → ICache
- Reset The Pipe Registers
- Generate Clock
- PLX Processor works autonomously