

**EE 6313**

**Advanced Microprocessor Systems**

**Project on**

**Design of 32-bit RISC microprocessor**

**Fall 2019**

**Done By**

**Kumar Ananda Chaya 1001671413**

**Surabhi Chythanya Kumar 1001678544**

**Chandrashekar Mohan 1001657329**

**CONTENTS**

|  |  |
| --- | --- |
| **TOPICS** | **PAGE NUMBERS** |
| INTRODUCTION | 3 |
| ALU INSTRUCTION SET | 4 |
| LD/ST INSTRUCTION SET | 6 |
| BRANCH INSTRUCTION | 9 |
| PIPELINE | 15 |
| INTERFACING | 26 |
| HAZARDS | 36 |
| EXTERNAL INTERRUPT HANDLING | 41 |

**INTRODUCTION:**

There are two types of microprocessor architecture namely, Reduced instruction set computer (RISC) architecture and complex instruction set computer (CISC) architecture.

As name indicates, instruction decoding in the CISC architecture will be complex and is mostly used for Application PC, laptop applications whereas there are relatively few instructions set in case of RISC architecture and is mostly used in smart phone processor architecture.

Our focus is to design a RISC architecture-based microprocessor.

**PROJECT OVERVIEW:**

The goal of this project is to design a 32-bit RISC microprocessor with load-store architecture with a 4-stage pipeline and Harvard architecture. The project also includes the instruction and data memory interfaces, register interface, and the entire pipeline control logic including full resolution of all structural, control, and data hazards.

The memory interface will only support a single asynchronous memory interface.

**SPECIFICATIONS:**

* Support 32-bit address bus, data bus and registers.
* All instructions in one fetch are 32-bit instructions.
* All instructions are aligned in 4N address.
* Harvard architecture constrained to one memory bus without cache.
* Supports 4 stages of pipeline.
* Thirty-two 32-bit registers are supported
* The memory space is provided with a single interface with A31..A2+BE3..0 addressing.
* The processor is of little-endian byte order.
* SP pointer convention that we have chosen as Pre decrement for PUSH and Post increment for POP

**ALU INSTRUCTION SET**

12

0

11

16

17

21

22

26

27

31

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OPCODE | DEST(RA) | SRC B(RB) | SRC C(RC) | K12 |

* OPCODE is 5bits length(Bit 31-27).
* DEST(RA) is the destination register, SRC B(RB) and SRC C(RC) are the source registers which are of 5 bit length(Bit 26-12).
* K12 is an offset of 12 bits(Bit 11-0).
* All the registers are 32 bits.
* ALU operations are only between the registers.

OPERATION:

1. MOV

Ex: MOV RA, RB – Moves contents of Reg B to Reg A

Here, RC=31 (escape code) and K12=0

1. ADD

Ex: ADD RA, RB, RC – Adds the contents of Reg B and C and the result is stored in Reg A

Here K12=0.

1. SUB

Ex: SUB RA, RB, RC – Adds the contents of Reg B and C and the result is stored in Reg A

Here K12=0.

1. NEG

Ex: NEG RA, RB - Negates the contents of Reg B and stores the result is Reg A

1. AND

Ex: AND RA, RB, RC – Performs AND operation of contents of Reg B and Reg C and stores the result in Reg A

1. OR

Ex: OR RA, RB, RC – Performs OR operation of contents of Reg B and Reg C and stores the result in Reg A

1. XOR

Ex: XOR RA, RB, RC – Performs XOR operation of contents of Reg B and Reg C and stores the result in Reg A

1. NOT

Ex: NOT RA, RB – Performs 1’s compliment operation of the contents of Reg B and stores in Reg A.

1. ROL

Ex: ROL RA, RB, RC – Performs circular left shift operation of contents of Reg B and contents of Reg C tells the number of times the bits of Reg B to be left shifted and stores the result in Reg A

1. ROR

Ex: ROR RA, RB, RC – Performs circular right shift operation of contents of Reg B and contents of Reg C tells the number of times the bits of Reg B to be right shifted and stores the result in Reg A.

1. ASR

Ex: ASR RA, #4, RB - Performs arithmetic right shift operation (used for signed number) of contents of Reg B i.e., #4 and contents of Reg C(#4) tells the number of times the bits of Reg B to be right shifted and stores the result in Reg A.

1. LSR

Ex: LSR RA, #2, RB - Performs logical right shift operation (used for unsigned number) of contents of Reg B i.e., #2 and contents of Reg C(#2) tells the number of times the bits of Reg B to be right shifted and stores the result in Reg A.

1. ASL/SHL

Ex: LSR RA, #3, RB – Performs arithmetic/logical right left operation of contents of Reg B and contents of Reg C(#2) tells the number of times the bits of Reg B to be right shifted and stores the result in Reg A.

1. MUL

Ex: MUL RA, RB, RC – Multiplies the contents of Reg B and Reg C and stores the result in Reg A

1. NOP

It has no operation and it increments PC by 4.

**LD/ST INSTRUCTION SET**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | DEST(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

* OPCODE is 5bits length (Bit 31-27).
* DEST(RA) is the destination register, SRC B(RB) and SRC C(RC) are the source registers which are of 5bit length (Bit 26-12).
* Option field (OPT) is a 2bit field (Bit 11-10).
* K10 is a signed offset field of 10 bits (Bit 9-0).
* All the registers are 32 bits.
* Memory operations are done by LD/ST.

OPT Field

|  |  |  |
| --- | --- | --- |
| 0 | NOP | No operation |
| 1 | PUSH/POP | Reg B +/-4 |
| 2 | DEC | Reg C -1 |
| 3 | INC | Reg C +1 |

LD/ST operation

For LD, RA,[RB+2N\*RC+K10]

For ST, [RB+2N\*RC+K10], RA

This is indirect, indexed, offset addressing mode.

Consider C code

int16\_t y[100];

i=100;

y[--i]=0;

Equivalent Assembly code is:

MOV RA, #0

MOV RB, &y[100] //RB is the base register

MOV RC, i //RC is decrement register (according to our convention)

ST [RB+2\*RC], RA

**MAPPING OF ABOVE INSTRUCTION**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | SRC(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10101 | #0 | &y[100] | i | 10 | 0000000000 |

The above example is indirect indexed addressing mode. Here offset is zero

Consider C code

int16\_t y[100];

Equivalent Assembly code is:

MOV RB, &y[0] //RB is the base register

MOV RC, i //RC is increment register (according to our convention)

LD RA, [RB+2\*RC]

**MAPPING OF ABOVE INSTRUCTION**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | DEST(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10000 | RA | &y[0] | i | 11 | 0000000000 |

The above example is indirect indexed addressing mode. Here offset is zero

1. LD32

Ex: LD32 RA,[RB+4\*RC+#12]- Loads 32 bit value from memory to reg A

1. LDU16

Ex: LDU16 RA,[RB+2\*RC+#12]- Loads unsigned 16 bit value from memory to reg A

1. LDS16

Ex: LDS16 RA,[RB+2\*RC+#12]- Loads signed 16 bit value from memory to reg A

1. LDU8

Ex: LDU8 RA,[RB+2\*RC+#12]- Loads unsigned 8 bit value from memory to reg A

1. LDS8

Ex: LDS8 RA,[RB+2\*RC+#12]- Loads signed 8 bit value from memory to reg A

**MAPPING OF LD32 INSTRUCTION**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | DEST(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10000 | RA | RB | RC | 00 | 0000001100 |

LD32 RA,[RB+4\*RC+#12]- Loads 32 bit value from memory to reg A

* Here OPCODE is (16)10
* OPT field is 00 which indicates there’s no special operation.
* From the above example, K10 offset field is (12)10

1. ST32

Ex: ST32 [RB+4\*RC+K10], RA Stores 32 bit value from Reg A to memory

1. ST16

Ex: ST16 [RB+2\*RC+K10], RA Stores 16 bit value from reg A to memory

1. ST8

Ex: ST8 [RB+RC+K10], RA Stores 8 bit value from reg A to memory

**MAPPING OF ST32 INSTRUCTION**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | SRC A(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10101 | RA | RB | RC | 00 | 0000001100 |

ST32 [RB+4\*RC+#12], RA Stores 32 bit value from Reg A to memory

* Here OPCODE is (21)10
* OPT field is 00 which indicates there’s no special operation.
* From the above example, K10 offset field is (12)10

**MAPPING OF PUSH INSTRUCTION**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | SRC A(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10101 | RA | 11100 | 11111 | 01 | 0000000100(+4) |

PUSH RA -> The SP is Pre decremented by 4 bytes and stores the 32-bit value from Reg A to stack memory

* PUSH operation is like store operation. Hence, OPCODE is (21)10
* RB is the stack pointer registers
* RC is 31 which is escape value since it is not used.
* OPT field is 01 which indicates the PUSH.
* From the above example, K10 offset field is +4.

**MAPPING OF POP INSTRUCTION**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | SRC A(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10000 | RA | 11100 | 11111 | 01 | 0000000000 |

POP RA -> The SP is Post incremented by 4 bytes and stores the 32-bit value from stack memory to Reg A.

* POP operation is like load operation. Hence, OPCODE is (16)10
* RB is the stack pointer registers
* RC is 31 which is escape value since it is not used.
* OPT field is 01 which indicates the POP.
* From the above example, K10 offset field is 0.

**BRANCH INSTRUCTIONS**

**OPERATION:**

1. GOTO {LABEL}

Ex: GOTO #0x20000000.

|  |  |
| --- | --- |
| 1000 0000 | GOTO |
| 1000 0004 | 20000000 |

Here PC will be loaded with the absolute 32-bit value.

From the above example, OPCODE GOTO will occupy 5 bits so the 32-bit value won’t fit in the remaining 27 bits. Hence, the absolute 32-bit value is stored in next memory location (next instruction).

PC←abs32

The above operation is equivalent to LD32 PC,[PC]

**MAPPING OF GOTO INSTRUCTION**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | DEST(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 11000 | 11110 | 11110 | 11111 | 00 | 0000000000 |

GOTO #0x20000000- Loads 32 bit value from memory to reg A

* Here GOTO is like load operation (LD32 PC,[PC]). Hence, OPCODE is (16)10
* RA and RB are PC registers whose number is 30.
* OPT field is 00 which indicates there’s no special operation.
* From the above example, K10 offset field is (12)10

1. CALL {LABEL}

Ex: CALL #0x20000000.

|  |  |
| --- | --- |
| 1000 0000 | CALL |
| 1000 0004 | 20000000 |
| 1000 0008 | *“Returns here*” |

Here Initially, value of PC will be saved in PCT i.e., temporary PC and then PC will be loaded with the absolute 32-bit value.

The PC is incremented by 4bytes and it stores the contents to PCT (save PC). Then PC gets the next instruction that is the absolute 32-bit value.

PCT←PC+4

PC←abs32

To load PCT, we must add a custom logic.

The above operation is equivalent to LD32 PC,[PC]

**MAPPING OF CALL INSTRUCTION**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | DEST(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 11001 | 11110 | 11110 | 11111 | 00 | 0000000000 |

CALL #0x20000000- Loads 32 bit value from memory to reg A

* Here CALL is like load operation (LD32 PC,[PC]). Hence, OPCODE is (16)10
* RA and RB are PC registers whose number is 30.
* OPT field is 00 which indicates there’s no special operation.
* From the above example, K10 offset field will be zero

1. RETURN

Here PC gets PCT

PC←PCT

To execute this instruction, we need to add custom logic.

**MAPPING OF RETURN INSTRUCTION**

0

31

11

12

16

17

21

22

26

27

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OPCODE | DEST(RA) | SRC B(RB) | SRC C(RC) | K12 |

0

31

11

12

16

17

21

22

26

27

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11010 | ? | ? | ? | ? |

1. INT N

Ex: INT #12.

Here Initially, value of PC will be saved in PCI then FLAGS will be pushed to FLAGSI. Then PC will be loaded with respective interrupt vector address.

|  |
| --- |
| PCI[N]←PC |
| FLAGSI[N]←FLAGS |
| PC←ivt[N] |

To load PCI and FLAGSI, we need to add a custom logic.

The above operation is equivalent to LD32 PC,[BASE+N\*4]

**MAPPING OF INT INSTRUCTION**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | DEST(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 11011 | 11110 | 11010 | 11111 | 00 | 0000001100 |

INT #12

* Hence, OPCODE is (16)10  (LD32 PC,[BASE+N\*4]).
* RA is PC (30).
* RB is base register (26).
* RC is escape value 31.
* OPT field is 00 which indicates there’s no special operation.
* From the above example, K10 offset field will be #12.
* PCI[N] and FLAGSI[N] are fast registers and are only in WB stage of pipeline. These can be written into WB and read over WB and there are no extra pipeline signals
* No access to PCI and FLAGSI

1. RETI

Here Initially, flags will be Popped and the PC is PC will be restored by PCI.

|  |
| --- |
| FLAGS←FLAGSI[N] |
| PC←PCI[N] |

To load PC and FLAGS, we need to add a custom logic.

The above operation is equivalent to LD32 PC,[BASE+N\*4](?)

**MAPPING OF RETI INSTRUCTION**

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OPCODE | DEST(RA) | SRC B(RB) | SRC C(RC) | OPT | K10 |

0

31

9

10

11

12

16

17

21

22

26

27

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 11100 | ? | ? | ? | ? | ? |

INT #12

* Hence, OPCODE is (16)10  (LD32 PC,[BASE+N\*4]).
* RA is PC (30).
* RB is base register (26).
* RC is escape value 31.
* OPT field is 00 which indicates there’s no special operation.
* From the above example, K10 offset field will be #12.

1. BRA rel27

Ex: BRA #8.

|  |  |
| --- | --- |
| BRA | #8 |

Here PC will be loaded with the PC and relative offset.

PC←PC + rel27\*4

We need to add custom logic to the above operation

**MAPPING OF BRA INSTRUCTION**

0

31

26

27

|  |  |
| --- | --- |
| OPCODE | OFS27 |

0

31

27

|  |  |
| --- | --- |
| 11101 | 000 0000 0000 0000 0000 0000 1000 |

The branch instruction can jump in the range of +/-228 or +/- 256MB jump

1. B cond rel23

Ex: BRA Z #24.

|  |  |  |
| --- | --- | --- |
| BRA | cond | #24 |

When if condition is true PC will be loaded with the PC and relative offset.

If(cond),PC←PC + rel27\*4

Custom logic is needed added for the above operation

**MAPPING OF BRA Z INSTRUCTION**

0

31

22

23

26

27

|  |  |  |
| --- | --- | --- |
| OPCODE | COND | OFS23 |

0

31

22

23

26

27

|  |  |  |
| --- | --- | --- |
| 11110 | cond | 000 0000 0000 0000 0001 1000 |

The conditional branch instruction can jump in the range of +/-224 or +/- 16MB jump

1. RCALL rel27

Ex: RCALL #8.

|  |  |
| --- | --- |
| RCALL | #8 |

Here PC will be loaded with the PCT (save PC) and PC will be loaded with relative offset.

PCT←PC

PC←PC + rel27\*4

Custom logic is needed for the above operation

**MAPPING OF RCALL INSTRUCTION**

0

31

26

27

|  |  |
| --- | --- |
| OPCODE | OFS27 |

0

31

27

|  |  |
| --- | --- |
| 11111 | 000 0000 0000 0000 0000 0000 1000 |

**PIPELINE**

There are 4 stages of pipeline.

1. Instruction Fetch
2. Read Register/Address Generation
3. Execute/ Fetch Operands
4. Write Back

Memory Interface

Register Interface

DFF

DFF

DFF

Execute/ Fetch Operands

Read Register/ Address Generation

Instruction Fetch

Write Back

Flush

Stall

1. **INSTRUCTION FETCH (IF)**

This is the 1st stage of pipeline. It fetches next instructions.

PC\_INIT

XACT\_IF\_ADD

XACT\_IF\_DATA

0 Reset

1 Normal

2 Write PC/Flush

32

0

PC

+4

PC Fetch

* When reset, the PC Fetch is loaded with PC\_INIT which is reset vector address. Zero is the address of the first instruction to be executed. This can be written as: LD PC,[#0].
* During normal operation, PC Fetch will be incremented by 4.
* When write PC/Flush, the PC loads the new value to PC Fetch.

IF Operation:

PCF 0 4 8 12 16 24 28 32 36 40

PC 0 0 0 4 4 24 24 24 28 28

Branch

Flush

During normal operation, PC Fetch will be incrementing by 4. During branch instruction, PC will be loaded with new value, and then it writes to PC Fetch.

**Signal generated from IF**

OPCODE IR[31:27]

OPT IR[11:10] if(OPCODE = LD/ST|PUSH |POP)

K10 IR[9:0] if(OPCODE = LD/ST|PUSH|POP)

K12 IR[11:0] if(OPCODE = ALU)

COND IR[26:23] if(OPCODE = BRA COND)

N IR[9:2] if(OPCODE = INT)

RD\_REGA\_EN = 1 if(OPCODE != RET|RETI|BRA|

BRA COND|ALU)

RD\_REGA\_NUM IR[26:22] if(RD\_REGA\_EN)

RD\_REGB\_EN = 1 if(OPCODE != RETRETI|BRA|BRA COND)

RD\_REGB\_NUM IR[21:17] if(RD\_REGB\_EN)

RD\_REGC\_EN = 1 if(OPCODE != RET|RETI|BRA|BRA COND)

RD\_REGC\_NUM IR[16:12] if(RD\_REGC\_EN)

RD\_MEM\_EN = 1 if( OPCODE = POP|LD)

WR\_MEM\_EN = 1 if( OPCODE = PUSH|ST)

WR\_REG\_EN = 1 if( OPCODE! = POP|LD)

WR\_REG\_NUM if( WR\_REG\_EN)

WR\_REG\_DATA if( WR\_REG\_EN)

WR\_INCDEC\_REG\_EN = 1 if( OPCODE = LD|ST && OPT =2|OPT =3)

WR\_INCDEC\_REG\_NUM if(WR\_INCDEC\_REG\_EN)

WR\_SP\_EN = 1 if( OPCODE = PUSH|POP)

WR\_PCT\_EN = 1 if( OPCODE = CALL | RCALL)

WR\_PC\_EN=1 if( OPCODE = BRA|BRA COND|RETI|

INT|CALL|RESET)

WR\_FLAG\_EN = 1 if( OPCODE = RETI)

1. **READ REGISTER/ ADDRESS GENERATION (RR/ADGEN)**

RR part of RR/ADGEN

RR is used to get ALU arguments.

1. Signals to Register interface

RI\_RD\_REGA\_EN = SRC A != 31

RI\_RD\_REGA\_NUM = SRC A

RI\_RD\_REGB\_EN = SRC B != 31

RI\_RD\_REGB\_NUM = SRC B

RI\_RD\_REGC\_EN = SRC C != 31

RI\_RD\_REGC\_NUM = SRC C

* SRC A,B and C are the signals from IF stage.
* In this stage, the data dependency hazard can be detected.

1. Signals from Register interface

RI\_RD\_REGA\_DATA – This can be SRC A for ST or DEST for ALU

RI\_RD\_REGB\_DATA – This can be SRC B for ADGEN or ARG1 for ALU

RI\_RD\_REGC\_DATA – This can be SRC C for ADGEN or ARG2 for ALU

ARGUMENT GERATION TO EX STAGE

RI\_RD\_REGC\_DATA

RI\_RD\_REGB\_DATA

ARG1\_DATA

(TO EX)

ARG2\_DATA

(TO EX)

K12

K12

Fn(SRC C)

Fn(SRC B)

**Signals generated from RR**

RD\_REGB\_NUM|RR

ALU\_ARG1

If(REGB\_EN|RR&&opcode=ALU)

RD\_REGC\_NUM|RR

ALU\_ARG2

If(REGC\_EN|RR&&opcode=ALU)

RD\_REGB\_EN|RR

RD\_REGB\_NUM|RR&&opcode>15

RI\_RD\_REGB\_DATA

RI\_RD\_REGC\_DATA

RD\_REGC\_EN|RR

RD\_REGC\_NUM|RR&&opcode>15

RI\_RD\_REGA\_DATA

RD\_REGA\_EN|RR

RD\_REGA\_NUM|RR&&opcode=ST|PUSH

If(opcode=POP|PUSH)

WR\_SP\_DATA = SP+K10

RD\_REGB\_EN

If(RD\_REGB\_NUM=27)

WR\_PCT\_DATA

If(CALL|RCALL)

If(opcode=GOTO|CALL)

WR\_PC\_DATA = abs32

WR\_PC\_DATA = PCT|PCI

If(opcode=RET|RETI)

WR\_PC\_DATA=PC\_DATA + OFFSET

If(BRA|BRACOND|RCALL)

WR\_PC\_DATA=IVT[N]

If(opcode = INT N)

WR\_PC\_DATA=reset vector address = 0

If(RESET)

ADGEN part of RR/ADGEN

ADGEN is used to generate memory address for LD/ST operation

RI\_RD\_REGB\_DATA

ADDRESS

RW\_MEM\_ADD

SRC B

0

SRC B == 31

RI\_RD\_REGC\_DATA

SRC C

<<

SRC C == 31

K10

0

N

[SRCB + SRCC<<2+SIGNED(K10)]

* Some of the signals are denoted different in further pipelines stages but they are of the same function.

Ex: RI\_RD\_REG\_DATA in RR stage becomes WR\_REG\_DATA in FO stage.

**OVERVIEW**

|  |  |  |
| --- | --- | --- |
| **INPUTS** | **OUTPUTS** | **PASS THROUGH** |
| OPCODE | ALU\_ARG1 | RD\_MEM\_EN |
| OPT | ALU\_ARG2 | WR\_MEM\_EN |
| K10 | RI\_RD\_REGA\_DATA | WR\_REG\_EN |
| K12 | RI\_RD\_REGB\_DATA | WR\_REG\_NUM |
| N | RI\_RD\_REGC\_DATA | WR\_INCDEC\_REG\_EN |
| RD\_REGA\_NUM | WR\_PC\_DATA | WR\_INCDEC\_REG\_NUM |
| RD\_REGB\_NUM | WR\_PCT\_DATA | WR\_SP\_EN |
| RD\_REGC\_NUM | WR\_FLAG\_DATA | WR\_PCT\_EN |
| RD\_REGA\_EN | RW\_MEM\_ADD | WR\_PC\_EN |
| RD\_REGB\_EN | WR\_SP\_DATA | WR\_FLAG\_EN |
| RD\_REGC\_EN |  |  |
| COND |  |  |

1. **FETCH OPERAND/EXECUTION (FO/EX)**

32

EX part of FO/EX

FLAG\_DATA

(TO WB)

ARG1\_DATA

(FROM RR)

32

5

32

32

32

ALU DATA

DONE\_EX

WR\_REG\_DATA

(TO WB)

Fn (OPCODE)

LD DATA

FROM FO

OPCODE

(FROM RR)

ARG2\_DATA

(FROM RR)

* ALU gets the operands from the RR stage and perform ALU operations depending on the OPCODE.
* The MUX selects the ALU data or LD data depending on the OPCODE logic which generates the WR\_REG\_DATA.

FO part of FO/EX

MEMORY INTERFACE

LD\_DATA

XACT\_FO\_ADD

XACT\_FO\_REQ

XACT\_FO\_DATA

XACT\_FO\_DONE

DONE\_FO

RD\_MEM\_EN

RD\_MEM\_ADD

RR

D Q

The LOAD DATA is generated from memory interface and DONE\_FO is obtained as soon as LOAD\_DATA is given from memory.

**Signals generated from FO/EX stage**

WR\_MEM\_DATA

RI\_RD\_REGA\_DATA

RI\_RD\_REGA\_DATA will become WR\_MEM\_DATA as it will be written into memory in WB stage.

RD\_MEM\_EN

WR\_REG\_DATA

RW\_MEM\_ADD

WR\_INCDEC\_REG\_DATA

RI\_RD\_REGC\_DATA

RI\_RD\_REGC\_DATA will become WR\_INCDEC\_REG\_DATA as it will be written into register in WB stage.

flag output

WR\_FLAG\_DATA

Flag output will become WR\_FLAG\_DATA as it will be written into flag in WB stage

DONE SIGNAL GENERATION

DONE\_EX

OP={ALU}

DONE\_FO

RD\_MEM\_EN

DONE\_FO

**OVERVIEW**

|  |  |  |
| --- | --- | --- |
| **INPUTS** | **OUTPUTS** | **PASS THROUGH** |
| RI\_RD\_REGA\_DATA | WR\_MEM\_DATA | WR\_PCT\_DATA |
| RD\_MEM\_EN | WR\_FLAG\_DATA | WR\_MEM\_EN |
| ALU\_ARG1 | WR\_REG\_DATA | WR\_SP\_EN |
| ALU\_ARG2 | WR\_INCDEC\_DATA | WR\_SP\_DATA |
| RI\_RD\_REGC\_DATA |  | WR\_PCT\_EN |
| RW\_MEM\_ADD |  | WR\_PCT\_DATA |
| OPCODE |  | WR\_PC\_EN |
| OPT |  | WR\_PC\_DATA |
| COND |  | WR\_REG\_EN |
| N |  | WR\_REG\_NUM |
|  |  | WR\_INCEDEC\_REG\_EN |
|  |  | WR\_INCEDEC\_REG\_NUM |
|  |  | WR\_FLAG\_EN |

1. **WRITE BACK (WB)**

This is the last stage in the pipeline and it writes to registers and memory.

INPUTS TO WB STAGE OF PIPELINE

**Write to registers signals**

WR\_REG\_EN (1 b)

WR\_REG\_NUM (5 b)

WR\_REG\_DATA (32 b)

**Write to memory signals**

WR\_MEM\_EN (1 b)

WR\_MEM\_ADD (32 b)

WR\_MEM\_DATA (32 b)

**Write to PC signals**

WR\_PC\_EN (1 b)

WR\_PC\_DATA (32 b)

**Write to flags signals**

WR\_FLAGS\_EN (1 b)

WR\_FLAGS\_DATA (32 b)

**Write to SP signals**

WR\_SP\_EN (1 b)

WR\_SP\_DATA (32 b)

**Increment and decrement registers**

INC\_DEC\_REG\_EN (1 b)

INC\_DEC\_REG\_NUM (5 b)

INC\_DEC\_REG\_DATA (32 b)

**Write to PCT signals**

WR\_PCT\_EN (1 b)

WR\_PCT\_DATA (32 b)

**INTERFACE TO MEMORY**

XACT\_WB\_REQ (1 b)

XACT\_WB\_ADD (32 b)

XACT\_WB\_DATA (32 b)

XACT\_WB\_DONE (1 b)

**INTERFACE TO REGISTERS**

RI\_WR\_REGA\_EN (1 b)

RI\_WR\_REGA\_NUM (5 b)

RI\_WR\_REGA\_DATA (32 b)

RI\_WR\_REGB\_EN (1 b)

RI\_WR\_REGB\_NUM (5 b)

RI\_WR\_REGB\_DATA (32 b)

RI\_WR\_SP\_EN (1 b)

RI\_WR\_SP\_DATA (32 b)

RI\_WR\_PC\_EN (1 b)

RI\_WR\_PC\_DATA (32 b)

RI\_WR\_PCT\_EN (1 b)

RI\_WR\_PCT\_DATA (32 b)

RI\_WR\_PCT\_EN (1 b)

RI\_WR\_PCT\_DATA (32 b)

**FEEDBACK SIGNAL**

DONE\_WB (1 b)

FLUSH (1 b)

* In WB it the writes to registers and memory and then PC is incremented by 4.
* During branch, WB writes to PC and sends the FLUSH signal.
* After write operation, DONE\_WB signal is generated.
* RI\_{}\_EN signal detects data hazards.

SIGNAL FLOW IN WB STAGE

Write to the register

WR\_REG\_NUM

WR\_REG\_EN

WR\_REG\_DATA

WR\_MEM\_EN

Write to the memory

RW\_MEM\_ADD

WR\_MEM\_DATA

WR\_INCDEC\_REG\_EN

WR\_INCDEC\_REG\_NUM

Write to inc/dec register

WR\_INCDEC\_REG\_DATA

WR\_FLAG\_EN

Write to flag register

WR\_REG\_NUM=26

WR\_FLAG\_DATA

WR\_SP\_EN

WR\_REG\_NUM=28

Write to SP register

WR\_SP\_DATA

WR\_PC\_EN

Write to PC register

WR\_REG\_NUM=30

WR\_PC\_DATA

WR\_PCT\_EN

WR\_PCT\_DATA

WR\_REG\_NUM=27

Write to PCT register

* Explicit write has highest priority over normal functionality.

Suppose if we write to FLAGS, the normal operation result of FLAGS is overwritten with the write operation result.

WR\_FLAG\_EN

WR\_FLAG\_DATA

NOT(WR\_REG\_EN&WR\_REG\_NUM==FLAGS)

Suppose if we write to SP, the normal operation result of SP is overwritten with the write operation result.

Ex : POP SP

WR\_SP\_EN

WR\_SP\_DATA

NOT(WR\_REG\_EN&WR\_REG\_NUM==SP)

Suppose if we write to PC, the normal operation result of PC is overwritten with the write operation result.

Ex : BRA COND

WR\_PC\_EN

WR\_PC\_DATA

NOT(WR\_REG\_EN&WR\_REG\_NUM==PC)

**INTERFACING**

**DESIGN OF EXTERNAL MEMORY INTERFACE**

32

XACT\_{ }\_EN

ADD

ADD

32

DATA

DATA

XACT\_{ }\_ADD

XACT\_{ }\_DATA

RD

RD

XACT\_{ }\_DIR(R/W)

WR

WR

READY

XACT\_{ }\_DONE

* { } can be any of the 3 stages: IF, FO and WB.
* XACT\_{ }\_DIR indicates direction for read and write operation.

PRIORITY

HIGHER PRIORITY

IF

FO

WB

IF\_EN

EN

PRIORITY ENCODER

FO\_EN

EN

WB\_EN

EN

* WB stage always has the highest priority.
* IF stage is always hardwired to zero.

ADDRESS SELECTION

XACT\_IF\_ADD

XACT\_FO\_ADD

ADD

XACT\_WB\_ADD

32

3

{ } \_ EN

DIR GENERATION

DIR

XACT\_WB\_DIR

XACT\_IF\_DIR

XACT\_FO\_DIR

{ } \_ EN

3

* The Address and DIR signals are selected by enable signals that are selected by priority encoder.
* For read operation (IF, FO), DIR=1 and for write operation (WB), DIR=0.

**Truth table for priority encoder**

|  |  |  |  |
| --- | --- | --- | --- |
| **I2** | **I1** | **I0** | **STAGE** |
| X | X | 1 | WB |
| X | 1 | 0 | FO |
| 1 | 0 | 0 | IF |

GENERATION OF DONE SIGNAL

IF\_EN

READY

IF\_DONE

FO\_EN

FO\_DONE

READY

WB\_EN

READY

WB\_DONE

* Significance of DONE signal: As soon as any of the stage of the pipeline completes its operations, the respective stage sends a DONE signal in order to avoid overwriting.
* There is no DONE signal for RR stage of pipeline as it is only between registers and is fast.

DATA SELECTION

XACT\_IF\_DATA

DATA

XACT\_FO\_DATA

XACT\_WB\_DATA

* DIR signal indicates the direction of dataflow.
* To facilitate bidirectionality, transceiver is used.
* IF, FO and WB EN signals selects one of the stages according to the priority.

TIMING DIAGRAM

ADDRESS

DATA

READY

RD

DIR

WR

* Bus cycle runs 4 times faster than instruction cycle.
* Clock is stabilized by Phase Lock Loop.
* Read and write is based on DIR.

**GENERATION OF MEMORY INTERFACE SIGNALS**

IF STAGE

MEMORY

XACT\_IF\_REQ=1

XACT\_IF\_DONE

XACT\_IF\_DATA

XACT\_IF\_R/~W=1

XACT\_IF\_S/~U=0

XACT\_IF\_SIZE=3

XACT\_IF\_ADD

PC

IF\_DATA

IF\_DONE

FO STAGE

MEMORY

XACT\_FO\_SIZE

XACT\_FO\_REQ

XACT\_FO\_ADD

XACT\_FO\_S/~U

XACT\_FO\_R/~W=1

XACT\_FO\_DATA

XACT\_FO\_DONE

FO\_DONE

RD\_MEM\_EN

RW\_MEM\_ADD

Fn(OPCODE)

Fn(OPCODE)

WR\_REG\_DATA

WB STAGE

MEMORY

XACT\_WB\_REQ

XACT\_WB\_S/~U

XACT\_WB\_SIZE

XACT\_WB\_R/~W=0

XACT\_WB\_DONE

XACT\_WB\_DATA

XACT\_WB\_ADD

WR\_MEM\_EN

RW\_MEM\_ADD

Fn(OPCODE)

Fn(OPCODE)

WB\_DONE

WR\_MEM\_DATA

|  |
| --- |
| D R0 Q |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13 |
| R14 |
| R15 |
| R16 |
| R17 |
| R18 |
| R19 |
| R20 |
| R21 |
| R22 |
| R23 |
| R24 |
| R25 |
| R26 |
| R27 |
| R28 |
| R29 |
| R30 |

**REGISTER INTERFACE**

32

WR\_REG\_DATA

32

REG\_DATA

RI\_{}\_DATA

32

SPECIAL SIGNALS

32

RI\_{}\_NUM

5

OE=0

* Registers R0-R25 are general purpose registers.
* Registers R26-R30 are special registers.

|  |  |
| --- | --- |
| **REGISTERS** | |
| 0-25 | General Purpose |
| 26 | BASE |
| 27 | PCT |
| 28 | SP |
| 29 | FLAGS |
| 30 | PC |
| 31 | escape code |

* WR\_REG\_DATA is normal write signal.
* REG\_DATA is for increment/decrement operations.
* Output enable is always on.

OE=0

WR\_REG\_DATA NOTHING R26 BASE

REG\_DATA WR\_PCT R27 PCT

WR\_FLAGS R28 FLAGS

WR\_SP R29 SP

WR\_PC R30 PC

**MEMORY INTERFACE**

Memory interface for ST operations

For example,

STB [R0], R1

* Let’s assume that XACT\_ADD=12345679h i.e., R0=12345679h
* Since it is a byte operation, size=1. Opcode tells the size
* Data is LSB aligned
* Memory data is a function of DATA, SIZE and ADDRESS.

MEM\_DATA= fn(DATA, SIZE, ADD1:0)

Where ADD1:0 are bank enable bits and upper 30 bits are address

|  |  |  |
| --- | --- | --- |
| MEM\_BE0 | 0 | 0 |
| MEM\_BE1 | 0 | 1 |
| MEM\_BE2 | 1 | 0 |
| MEM\_BE3 | 1 | 1 |

MEM\_BE=fn(ADD1:0, SIZE)

MEM\_ADD=ADD31:2

BE

BE3 1

BE2 1

BE1 0

BE0 1

DATA7:0

XX

XX

REGISTER

MEMORY

XX

DATA23:16

DATA15:8

DATA7:0

DATA31:24

* Since last 2 bits of R0 is 01(0x12345679), BE1 is selected.
* The rest of the bytes are ignored(xx).

**All switch combinations**

REGISTER

DATA31:24

DATA23:17

MEMORY

DATA7:0

DATA15:8

There are totally 9 possible combinations.

Memory interface for LD operations

For example,

LDSB R1, [ R0]

* Let’s assume that XACT\_ADD=12345679h i.e., R0=12345679h
* Since it is a byte operation, size=1. Opcode tells the size
* Data is LSB aligned
* All registers are 32bits wide. Hence, it must be signed or unsigned extended.
* Signed or unsigned extension depends on OPCODE.

REGISTER

MEMORY

XX

XX

XX

DATA7:0

DATA15:8

DATA23:17

DATA31:24

DATA15:8

* A byte of data is loaded to lower 8 bits of register and the rest of the bits are sign extended for this example.
* It is a function of SIGNED/UNSIGNED, SIZE, ADDRESS1:0

**All switch combinations**

REGISTER

MEMORY

S

S

Z

Z

Z

4:1

5:1

2:1

2:1

5:1

(SSSSSSSS)

(00000000)

DATA15:8

DATA23:16

DATA31:24

fn(S/U, SIZE,ADD1:0)

* The lower 8 bits always gets data from 4:1 mux connected from any one of memory locations.
* The remaining bits will get either signed or unsigned extended bits depending upon the opcode and size which is selected by mux.

Overview of LD and ST memory interface switch combinations

MEMORY

REGISTER (32b)

|  |
| --- |
| ST32 |
| ST16 |
| ST8 |
| LD32 |
| LDU16 |
| LDS16 |
| LDU8 |
| LDS8 |

D31

D0-31

D0-15, D8-23, D16-31

(SSSSSSSSSSSSSSSSSSSSSSSS), D8-15, D16-23, D16-31

(0000000000000000), D8-15, D16-23, D16-31

(SSSSSSSSSSSSSSSS), D8-15, D16-23, D16-31

(000000000000000000000000), D8-15, D16-23, D16-31

D0

D0-31

D0-7, D8-15, D16-23, D16-31

**HAZARDS**

TYPES OF HAZARDS

1. Structural Hazards
2. Control Hazards
3. Data Hazards

**STRUCTURAL HAZARDS**

It is caused due to the fault in the structure/architecture of microprocessor. This is because pipeline stages(IF, FO, WB) will be communicating with only one memory.

Later this is resolved using STALL logic.

**CONTROL HAZARDS**

This is caused due to change in the normal operation of PC like BRA,BRA COND, RETI, CALL, GOTO, Write PC instructions.

This can be resolved by FLUSH logic in the pipelines.

**DATA HAZARDS**

This caused to due data dependencies like Read After Write(RAW),Write After Read(WAR), Write After Write(WAW) conditions in instructions.

This can be resolved by data forwarding.

**DATA HAZARDS**

Consider an example,

MOV R0, Z

MOV R1, R0

In first instruction, R0 is being written. In second instruction, R0 is being read. But, when R0 is read, it is not yet been updated by new value which will be written in first instruction. Hence, wrong (old value) value of R0 will be read. So, 2nd instruction is dependent on 1st instruction to get new value (correct value). This is known as DATA DEPENDENCY. This condition is called Read After Write (RAW) Hazard.

RAW Hazard Detection

If ((RD\_REGX\_NUM|RR==WR\_REGX\_NUM|EX) && (RD\_REGX\_EN|RR=1 && WR\_REGX\_EN|EX=1))

Where Xϵ [A,B,C]

If the above condition is true, RAW is detected.

This can be resolved by delaying reading R0 by 2 methods

Method 1: Stall

In this method, we delay by adding bubble (STALL) to 2nd instruction. A bubble does not increment PC by 4. This is added prior to RR stage so that reading R0 gets delayed and by this time R0 gets updated with new value.

Instruction 1: F1 RR1 EX1 WB1

Instruction 2: F2 RR2 EX2 WB2

STALL

Method 2: Adding NOP

Adding NOP increments PC by 4 and delays reading R0.

Instruction 1: F1 RR1 EX1 WB1

Instruction N: NOP NOP NOP NOP

Instruction 2: F2 RR2 EX2 WB2

By using the above 2 methods, we are wasting clock cycles unnecessarily and also in method 2, it makes the program bigger and busier. Hence, we use DATA FORWARDING as a solution.

**DATA FORWARDING**

Once the hazard is detected in RR stage, ALU knows about the hazard and then R0 is routed to that data from ALU and forward R0 directly to RR2 in the same clock telling RR2 not to read from register. Thus, ALU result i.e., new value is directly forwarded to R0 in RR2.

In this way, no clocks are wasted.

Instruction 1: F1 RR1 EX1 WB1

DATA FORWARDING

Instruction 2: F2 RR2 EX2 WB2

The data can be forwarded from ALU, WRITE\_REG\_DATA(memory read), PC, PCT, SP, INC/DEC REG, FLAGS.

Case1: From ALU

If ((RD\_REGA\_NUM|RR==WR\_REGA\_NUM|EX) && (RD\_REGA\_EN|RR=1 && WR\_REGA\_EN|EX=1)) then WR\_REGA\_DATA=RD\_REGA\_DATA

If ((RD\_REGB\_NUM|RR==WR\_REGB\_NUM|EX) && (RD\_REGB\_EN|RR=1 && WR\_REGB\_EN|EX=1)) then WR\_REGB\_DATA=RD\_REGB\_DATA

If ((RD\_REGC\_NUM|RR==WR\_REGC\_NUM|EX) && (RD\_REGC\_EN|RR=1 && WR\_REGC\_EN|EX=1)) then WR\_REGC\_DATA=RD\_REGC\_DATA

Case2: WRITE\_REG\_DATA(memory read)

If ((RD\_REGA\_NUM|RR==WR\_REGA\_NUM|EX) && (RD\_REGA\_EN|RR=1 && WR\_REGA\_EN|EX=1)) then WR\_REGA\_DATA=WR\_MEM\_DATA

If ((RD\_REGB\_NUM|RR==WR\_REGB\_NUM|EX) && (RD\_REGB\_EN|RR=1 && WR\_REGB\_EN|EX=1)) then WR\_REGB\_DATA=WR\_MEM\_DATA

If ((RD\_REGC\_NUM|RR==WR\_REGC\_NUM|EX) && (RD\_REGC\_EN|RR=1 && WR\_REGC\_EN|EX=1)) then WR\_REGC\_DATA=WR\_MEM\_DATA

Case 3: PC

If ((RD\_REGA\_NUM|RR==WR\_REGA\_NUM|EX) && (RD\_REGA\_EN|RR=1 && WR\_PC\_EN|EX=1)) then WR\_REGA\_DATA=WR\_PC\_DATA

If ((RD\_REGB\_NUM|RR==WR\_REGB\_NUM|EX) && (RD\_REGB\_EN|RR=1 && WR\_PC\_EN|EX=1)) then WR\_REGB\_DATA=WR\_PC\_DATA

Case 4: PCT

If ((RD\_REGA\_NUM|RR==WR\_REGA\_NUM|EX) && (RD\_REGA\_EN|RR=1 && WR\_PCT\_EN|EX=1)) then WR\_REGA\_DATA=WR\_PCT\_DATA

If ((RD\_REGB\_NUM|RR==WR\_REGB\_NUM|EX) && (RD\_REGB\_EN|RR=1 && WR\_PCT\_EN|EX=1)) then WR\_REGB\_DATA=WR\_PCT\_DATA

Case 5: SP

If ((RD\_REGA\_NUM|RR==WR\_REGA\_NUM|EX) && (RD\_REGA\_EN|RR=1 && WR\_SP\_EN|EX=1)) then WR\_REGA\_DATA=WR\_SP\_DATA

If ((RD\_REGB\_NUM|RR==WR\_REGB\_NUM|EX) && (RD\_REGB\_EN|RR=1 && WR\_SP\_EN|EX=1)) then WR\_REGB\_DATA=WR\_SP\_DATA

Case 6: INC\_DEC\_REG

If ((RD\_REGC\_NUM|RR==WR\_REGC\_NUM|EX) && (RD\_REGC\_EN|RR=1 && WR\_INC\_DEC\_EN|EX=1)) then WR\_REGC\_DATA=WR\_INC\_DEC\_DATA

Case 7: FLAGS

If ((RD\_REGA\_NUM|RR==WR\_REGA\_NUM|EX) && (RD\_REGA\_EN|RR=1 && WR\_FLAGS\_EN|EX=1)) then WR\_REGA\_DATA=WR\_FLAGS\_DATA

If ((RD\_REGB\_NUM|RR==WR\_REGB\_NUM|EX) && (RD\_REGB\_EN|RR=1 && WR\_FLAGS\_EN|EX=1)) then WR\_REGB\_DATA=WR\_FLAGS\_DATA

**STRUCTURAL HAZARD**

To resolve this hazard, we use STALL logic.

**STALL**

It is used in between the pipeline stages. A stage can be STALLED due to Self-Stall (when it is not Done) or when next stage or previous stages are STALLED.

RR stage cannot Self Stall.

CLK

STALLN+1

EN

Q

D

STAGE N+1

STAGE N

INSERT\_BUBBLE= STALLN v FLUSH

**CASE1:** When STAGE N is STALLED and STAGE N+1 is NOT STALLED

When STAGE N is STALLED, it inserts a bubble by making STALLN=1. No signals are sent to STAGE N+1 as STAGE N is not Done.

**CASE2**: When STAGE N+1 is STALLED and STAGE N is NOT STALLED

When STAGE N+1 is STALLED, it sends STALLN+1=1 making EN signal 0. Thus turning off DQ flip flop and thus prevents taking the data from previous stage as STAGE N+1 is not Done. Since DQ flip flop is disabled the inputs to it will be don’t care state.

**CASE3:** When both the stages are STALLED

When both the Stages are STALLED, the DQ flip flop will be disabled by STAGE N+1 and STAGE N will insert bubble. Since DQ flip flop is disabled the inputs(bubble) to it will be don’t care state.

**TRUTH TABLE**

|  |  |  |  |
| --- | --- | --- | --- |
| **STALLN** | **STALLN+1** | **EN** | **BUBBLE** |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | X |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | X |

EN=STALLN+1

BUBBLE=STALLN

STALL=DONE=NOT\_DONE

**CONTROL HAZARD**

This can be resolved by FLUSH logic.

**FLUSH**

In this, bubble is sent to all stages and the value that was calculated in those stages are flushed.

Memory Interface

Register Interface

DFF

DFF

DFF

Execute/ Fetch Operands

Read Register/ Address Generation

Instruction Fetch

Write Back

Flush

Stall

Bubble

Bubble

Bubble

When we write to PC, WB stage sends a FLUSH signal to all the stage by inserting bubble and all the old values are FLUSHED. Then new PC will be fetched by IF stage and normal operation continues.

PC Fetch will also be FLUSHED.

(WR\_PC\_EN|WB) v (WR\_REG\_EN|WB ^ WR\_REG\_NUM|WR=PC) FLUSH=1

**EXTERNAL INTERRUPT HANDLING**

When interrupt instruction (INT ) is fetched by IF stage, we have to make sure that it is not flushed when PC is changed. So it has to be not Flushable.

If ((OPCODE=INT ) ^ ((WR\_PC\_EN|WB) v (WR\_REG\_EN|WB ^ WR\_REG\_NUM|WR=PC)

DO\_NOT\_FLUSH=1

Here, when any pipeline gets FLUSH and it has INT instruction, DO\_NOT\_FLUSH is made 1 i.e., not Flushable. Thus, it is not lost.