REPORT OF MIPSR3000

by Kevin LASTRA

The original authors of the architecture description are
D. Hommais and P. Bazargan Sabet

Rewrited on SystemC by
F. Pecheux

Last update on march 01 2022 by
K. Lastra

Contents

1 RESUME

This MIPS1(first verision of mips 1981) architecture resume is maid in part of the Project "SystemC modeling for pipelined RiscV and assembly of TME platform", its have for goal to help the students for the good understanding of basic mips components, tools and structures.

The systemC architecture description was updated to the last version at this day "systemc2.3.3"

2 MIPS STRUCTURE

This architecture is based on the mips1 and/or mips32 instruction set. The architecture is a 5 stage pipelined, each stage are compound by 3 elements:

1. The stage core witch is named like the stage, ex. decode.h
2. A multiplexer which is gonna handle the information flow in the pipeline, named like the stage with the "mux" prefix, ex. mux_decode.h.
3. A fifo (first in first out) which gonna literally just put the input on the output, named like the stage with the "ff" prefix, ex. ff_decode.h
4. Exception : the alu is defined on the core but directly connected to the stage execute.

2.1 PIPELINE

2.1.1 STRUCTURE

The pipeline is compound with the manager (multiplexer) and the fifo.

- A pipeline have 2 types of register:
  1. command register (ex. opcode)
  2. data register (ex. toper or soper)
- Each register have different behavior:
  1. The command register can do 2 actions:
     - Either keep the data (keep)
     - Either charge the new data (!keep)
  2. The data register can do 3 actions:
     - Either conserve the command (hold)
     - Either delete the command (bubble ex. Eret insert a nop*)
     - Either add a new command (!hold and !bubble)

2.1.2 INSTRUCTION FLOW CONTROL

Three cases can happen:

- Kill : the instruction in the corresponding stage is killed.
- Stall : the instruction is not allowed to pass to the next pipe stage.
- Copy : the instruction is duplicated. A copy remains in the current stage and the other goes down the pipe.
- Exec : the instruction can be executed.

Exec(execute signal) is equal to "not (Copy or Stall or Kill)", each stage have different situation for these 3 signals.

1. Instruction Fetch:
   - Copy, the instruction is never copied.
   - Stall, stalled if:
     - the next stage (Decode) is occupied.
     - the instruction memory is not able to answer the instruction fetch request.
   - Kill, killed if:
     - the third previous instruction causes an exception.
     - a hardware or software interrupt occurs.
     - the previous instruction is a sleep.
2. Decode:
   - Copy, the instruction in the decode stage is copied if the current instruction is a sleep.
   - Stall, stalled if:
     - the next stage (Execute) is occupied.
     - there is a data hazard that cannot be resolved by bypasses.
     - the instruction memory cannot answer the instruction fetch (the instruction cannot be executed because ir may change the instruction stream).
   - Kill, killed if:
     - the second previous instruction causes an exception.
     - a hardware reset is detected.
     - a hardware or a software interrupt occurs.
3. Execute:
   - Copy, the instruction is never copied.

- Stall, stalled if:
  - the next stage (Memory Access) is occupied.
  - there is a data hazard that cannot be resolved by bypasses.
- Kill, killed if:
  - the previous instruction causes an exception.
  - a hardware reset is detected.
  - a hardware or a software interrupt occurs.

4. Memory:
   - Copy, the instruction is copied if the instruction has a copying capability(that is, it is a swap instruction and is making its first access).
   - Stall, stalled if the data memory is not able to answer the request.
   - Kill, killed if:
     - it causes an exception.
     - a hardware reset is detected.
5. Write back, the instruction in write back is always executed.

These follows a summary of different situations.

	I	D	E	M	W
reset	K	K	K	K	E
exception	K	K	K	K	E
interrupt	K	K	K	E	E
I_FRZ	S	S	E	E	E
D_FRZ	S	S	S	S	E
hazard in DEC	S	S	E	E	E
hazard in EXE	S	S	S	E	E
SLEEP	K	C	E	E	E
SWAP - first access	S	S	S	C	E

Note that if more than one situation occur in the same time Kill is prior than Stall which is prior than Exec.

3 AGREEMENT FOR INPUT, OUTPUT OR INNER SIGNALS DECLARATION

The agreement for declare a wire is compound by a name or an acromnym, and a suffix connected by an underscore.
The suffix is compound by 2 capital letters, the wire type and the stage.

- TYPES:
  - S: signal (wire which is not connected to a register from a fifo)
  - R: register (wire which is connected to a register from a fifo)
  - X: exception signal
- STAGE:
  - I: ifetch
  - D: decod
  - E: exec
  - M: memory
  - W: writeback
- EXAMPLE:

  "RES_SE"
  name : resultat
  type : signal
  stage : exec

4 STAGES AND COMPONENTS DESCRIPTION

In addition of all components files, "constants.h" contains all instructions encoding and special signals encoding.

4.1 IFETCH

4.1.1 IFETCH COMPONENT

Ifetch only manage 2 thinks, a bool witch tell if then incoming instruction is a delayed slot and a signal of 32 bits witch define the status register for the incoming instruction.

4.1.2 IFETCH MUX

The mux is gonna manage the instruction register, the new instruction fetched, a bool witch tell if is or not a delayed slot, the pc register and the status register with the 3 base mux signal(BUBBLE, HOLD and KEEP).

4.1.3 IFETCH FIFO

The input of the fifo is the output of the mux.

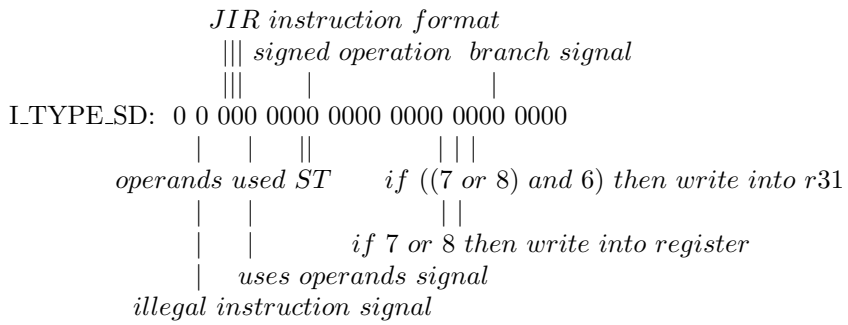
4.2 DECODE

Decod can handle 3 instruction format J, I and R.
Decod use the coprocessor 0 and handle exceptions.
HIGH and LOW register are used by the exception handler(for example for instruction emulation see index 5).
Multiple interesting signals:

- I_DUSE_SD: uses operands signal
- I_READS_SD: instruction uses S operands (soper)
- I_READT_SD: instruction uses T operands (toper)
- I_OSGND_SD: signed operation
- I_WRT31_SD: write into r31
- I_WRITE_SD: write into reg

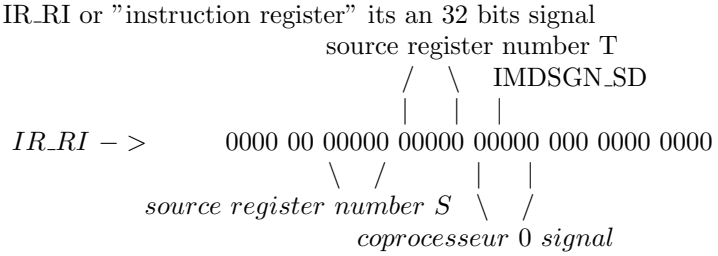
4.2.1 INSTRUCTION TYPE

I_TYPE_SD or "instruction type" its an 25 bits signal



4.2.2 INSTRUCTION REGISTER

IR_RI or "instruction register" its an 32 bits signal



dest reg number =	0x1F if write into r31
	IR_RI[15,11] if write into reg and R instruction format
	IR_RI[20,16] if write into reg and I instruction format
	else 0

COP0OP =	BADVADR if IR_RI[15,11] == badvadr_s
	NEXTSR if IR_RI[15,11] == status_s
	EPC if IR_RI[15,11] == epc_s
	CAUSE if IR_RI[15,11] == cause_s
	else 0

- badvadr_s → bad virtual adresse
- status_s → next instruction status register
- epc_s → exception pg counter reg
- cause_s → cause register

```
            (cop0_g ?)
COP0 = | cop0_g << 6 | IR_RI[22,21] << 3 | IR_RI[24,23] if IR_RI[25] == 0
       | else (cop0_g << 6) | 0x20 | IR_RI[4,0]

         | (special_g << 6) | IR_RI[5,0]     if IR_RI[31,26] == special_i
OPCOD = | (special_g << 5) | IR_RI[20,16]  if IR_RI[31,26] == bcond_i
         | COP0                             if IR_RI[31,26] == cop0_i
         | else (others_g << 6) | IR_RI[31,26]
```

4.2.3 PC AND BRANCH

```
IMDSEX = 0xFFFF    if IMDSGN && I_OSGND
         else 0x0
OFFSET = IMDSEX[13,0] << 18 | IR_RI[15,0] << 2

          | SOPER_SD if OPCOD == (jr_i or jalr_i)
NEXTPC = | JMPADR_SD if OPCOD == (j_i or jal_i)
          | BRAADR_SD if OPCOD is branch instruction and condition is valid
          | SEQADR_SD ex. (OPCOD == (beq_i) && S_EQ_T_SD == 1)
BRAADR = NEXTPC + OFFSET
SEQADR = NEXTPC + 4
JMPADR = JMPADR[31,28] . JMPADR[27,2] . 00
                  |              |
                  |              |
              NEXTPC[31,28]   IR_RI[25,0]

         | SEQADR      if OPCOD == bltzal_i or jalr_i or jal_i or bgezal_i
         | IR_RI[13,6]   if OPCOD == sra_i or srl_i or sll_i
IOPER = | IR_RI[15,0] << 16   if OPCOD == lui_i
         | LO_RW        if OPCOD == mflo_i
         | HI_RW        if OPCOD == mfhi_i
         | else IMDSEX_SD << 16 | IR_RI[15,0]
```

Decode manage 4 branch condition:

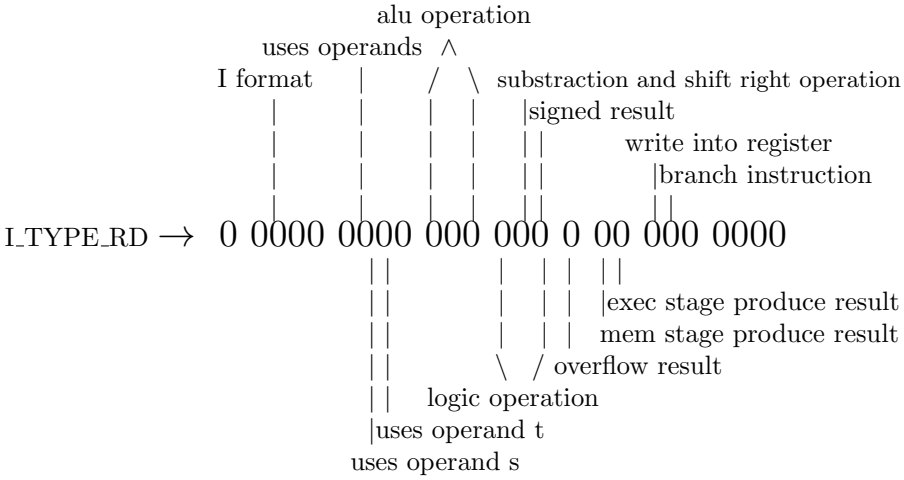
- S_CMP_T = SOPER xor TOPER compare condition
- S_EQ_T = (S_CMP_T == 0x0) equal condition
- S_LT_Z = (SOPER[31] == 1) less than zero condition
- S_LE_Z = (SOPER[31] == 1 or SOPER == 0x0) less or equal zero

4.3 EXECUTE

4.3.1 EXECUTE COMPONENT

4.3.1.1 INSTRUCTION TYPE

I_TYPE_RD ou "instruction type" c'est une signal sortant du pipeline a 25 bits



4.3.1.2 RESULTS

Exec produce 2 results:

- X or XOPER = IOPER if OPCOD == (sll_i or srl_i or sra_i) else X_SE
- Y or YOPER = IOPER if I_IFMT_SE == 1 else Y_SE

4.3.1.3 ERROR MANAGEMENT

WREDOPC is the signal redopc write enable, redopc is the adresse to return when jump to syscall or exception code.
WREDOPC = I_BRNCH_SE

IABUSER is the instruction adresse bus error is equal to (not I_BERR_N) witch is the signal of instruction bus error.

BREAK is the break signal
BREAK = (OPCOD == break_i)

SYSCALL is the syscall signal
SYSCALL = (OPCOD == syscall_i)

OVR is the overflow signal
OVR = OVERFLW_SE and I_OVRF_SE

IAMALGN is the instruction adresse miss alignement signal
IAMALGN = NEXTPC[1] or NEXTPC[0] -> the first bit and the second because pc is a multiple of 4 if not its means the adresse is non aligned

IASVIOL is the instruction adresse segmentation violation signal
IASVIOL = NEXTPC[31] and NEXTSR[3] if OPCOD == rfe_i else NEXTPC[31] & NEXTSR[1]

4.3.2 ALU

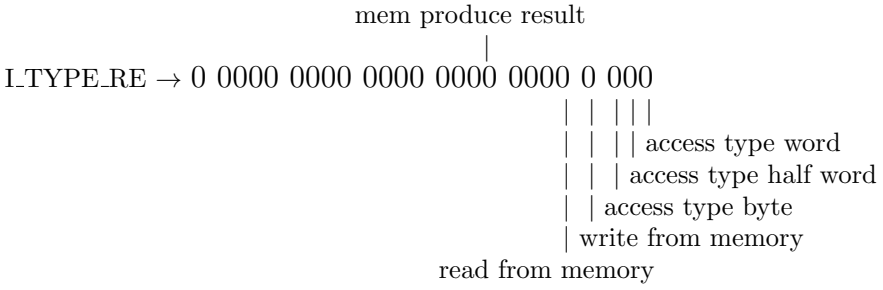
The ALU are made for make logique operations (AND, OR, NOR and XOR), arithmetique operations (addition and substraction), shift operations and slt compare operation.

The ALU component is instantiated on the core but connected directly with the execute stage.

4.4 MEMORY

4.4.1 INSTRUCTION TYPE

I_TYPE_RE ou "instruction type" c'est une signal sortant du pipeline a 25 bits.



I_WRITE_SM or "write into register" signal is equal to (I_TYPE[8] | I_TYPE[7])

4.4.2 MEMORY MANAGEMENT

DACCESS_SM or "data memory access" is the signal witch is gonna interact with the memory for read or write.
DACCESS = (I_STOR_SM or I_LOAD_SM)

WRITE_SM or "write into storage" is the signal witch for write into the memory.
WRITE_SM = I_STOR_SM and not FSTSWAP_SM. The signal FSTSWAP is equal to (SWAP and COPYCAP) but all the swapsignal are deprecated too thus the signal FSTSWAP or WRITE_SM need to be reworked.

DLOCK_SM or "lock data access" = FSTSWAP_SM, same problem of WRITE_SM (signal deprecated).
DATARED_SM or "read access" = (DACCESS_SM and not WRITE_SM).

RD_SM or "destination register" is equal to 00000 if (SWAP_RE and nop COPYCAP_RE) else RD_RE, SWAP signaldeprecated, RD_SM need to be rework.

temp = (I_BYTE_SM << 4) or (I_HALF_SM << 3) or (I_WORD_SM << 2) or RES_SE[1,0]

BYTSEL_SM =
| 0x1 if temp == 0x10 BYTSEL(byte select for read and write)
| 0x2 if temp == 0x11
| 0x4 if temp == 0x12
| 0x8 if temp == 0x13
| 0x3 if temp == 0x08
| 0xC if temp == 0x0A
| 0xF if temp == 0x04
| else 0x0

REDDAT_SM or "aligned data" =
| D_IN[31, 0] if BYTSEL[0] == 1
| D_IN[31, 8] + 0x00 if BYTSEL[1] == 1
| D_IN[31, 16] + 0x0000 if BYTSEL[2] == 1
| else D_IN[31, 24] + 0x000000

BSEXT_SM = 0xFFFFFF if (REDDAT[7] == 1 && OPCOD == lb_i) else 0x000000
HSEXT_SM = 0xFFFF if (REDDAT[15] == 1 && OPCOD == lh_i) else 0x0000

DATA_SM is the result of the bus data

$$\text{DATA\_SM} = \begin{cases} \text{REDDAT\_SM if (OPCOD == lw\_i) or (OPCOD == swap\_i)} \\ \text{BSEXT << 8 or REDDAT[7,0] if OPCOD == (lb\_i or lbu\_i)} \\ \text{HSEXT << 8 or REDDAT[15,0] if OPCOD == (lh\_i or lhu\_i)} \\ \text{else RES\_RE} \end{cases}$$

4.4.3 MEMORY ERRORS, EXCEPTIONS AND INTERRUPTIONS MANAGEMENT

DABUSER or "data adresse bus error" signal.
DABUSER = not D_BERR_N, D_BERR_N the bus error signal.

LAMALGN or "load adresse miss alignment".

$$\text{LAMALGN} = \begin{cases} \text{RES\_RE[1] or RES\_RE[0] if I\_WORD and I\_LOAD} \\ \text{RES\_RE[0] if I\_HALF and I\_LOAD} \\ 0 \end{cases}$$

SAMALGN or "store adresse miss alignment".

$$\text{SAMALGN} = \begin{cases} \text{RES\_RE[1] or RES\_RE[0] if I\_WORD and I\_STOR} \\ \text{RES\_RE[0] if I\_HALF and I\_STOR} \\ 0 \end{cases}$$

LASVIOL or "load adresse segmentation violation".

$$\text{LASVIOL} = \begin{cases} \text{RES\_RE[31] and SR\_RE[1] if I\_LOAD} \\ 0 \end{cases}$$

SASVIOL or "store adresse segmentation violation".

$$\text{SASVIOL} = \begin{cases} \text{RES\_RE[31] and SR\_RE[1] if I\_STOR} \\ 0 \end{cases}$$

BADDA or "bad data adresse" is equal to (SASVIOL or LASVIOL or LAMALGN or SAMALGN).

IASVIOL is the instruction adresse segmentation violation.
IAMALGN is the instruction adresse miss alignment.
BADIA or "bad instruction adresse" is equal to (IASVIOL or IAMALGN).

EXCCODE =
| 0 if INTRQ interrupt request
| 4 if LAMALGN or LASVIOL or IAMALGN or IASVIOL
| 5 if SAMALGN or SASVIOL
| 6 if IABUSER instruction adresse bus error
| 7 if DABUSER data adresse bus error
| 8 if SYSCALL syscall exception
| 9 if BREAK break exception
| 0xA if ILLGINS unknown instruction
| 0xB if C0UNUSE coprocessor 0 unused
| else 0xC

CAUSE_XM or "exception cause (exp)" signal of 32 bits

CAUSE_XM = x . 0 . xx . 0x000 . xxxxxx . xx . 00 . xxxx . 00

| | | | | | |
BDSLOT COPERR IT CAUSE_RX EXCCODE

CAUSE_SM or "exception cause (software)" signal of 32bits.

CAUSE_SM = xxxx xxxx xxxx xxxx . xxxxxx . xx . xxxx xxxx

| | | | |
CAUSE_RX[31,16] IT_XX RES_RE[9,8] CAUSE_RX[7,0]

WCAUSE_SM or "exception cause write enable (software)"
WCAUSE_SM = 1 if OPCOD == (mtc0_i or cause_s) else 0

LATEEX_XM or "late exceptions" = DABUSER_XM

EARLYEX_XM or "early exceptions" = ILLGINS or C0UNUSE or IAMALGN or IASVIOL or IABUSER or OVF or BREAK or SYSCALL or LAMALGN or LASVIOL or SAMALGN or SASVIOL

EXCRQ or "exception request" = EARLYEX or LATEEX

SWINTRQ or "software interrupt request" = CAUSE_RX[9,8]

SWINT_XM or "software interrupt (mtc0)" = RES_SE[9,8] if (OPCODE == (mtc0_i and cause_s) and not KEEP) else CAUSE_RX[9,8]

WEPC or "exception program counter write enable" = EXCRQ_XM

4.4.4 PC AND EPC

RSTORSR_SW or "next instruction status(rfe)"
RSTORSR_SW = NEXTSR[31,4] . NEXTSR[5,2]

NEXTSR_SM or "next instruction status (software)" = RSTORSR_SM ir OPCODE == rfe_i else RES_RE

WSR_SM or "next status write enable"
WSR_SM = 1 if (OPCOD == rfe_i) or (OPCOD == mtc0_i and COP0D == status_s) else 0

EPC_XM or "exception program counter" = PC_RE if BDSLOT == 0 else REDOPC

4.5 WRITE BACK

4.5.1 INSTRUCTION TYPE

I_TYPE_RM ou "instruction type" c'est une signal sortant du pipeline a 25 bits

I_WRITE_SM or "write into register" signal is equal to (I_TYPE[8] | I_TYPE[7])

4.5.2 HI AND LOW

WLO or "low register write enable" = 1 IF OPCOD == mtlo_i else 0

WLO or "high register write enable" = 1 IF OPCOD == mthi_i else 0

5 INSTRUCTION EMULATION

The instruction emulation is a mechanism which enables puts new assembly instructions not taken by the hardware but included in the exception handler list.

For example, we can create a multiplication program, add it to the exception handler and identify it with the opcode "mult", next to that we put the opcode "mult" where we need a multiplication and when the cpu decod de multiplication opcode("mult") the cpu is gonna generate an exception named "unknown instruction" then the cpu is gonna verify the exception handler and then he gonna find the mult handler witch is gonna branch to the "mult" fonction, do his job and write in the high and low register and then return to the instruction "mult" on de bade code.