UHasselt


Computer Architecture: RISC-V processor

# Een RV32I Implementatie

*Authors:*
Jonas Vannieuwenhuijsen
Jeroen Van Caekenberghe

26/1/2021

# Inhoudsopgave

# 1 Inleiding

In volgend verslag bespreken wij de werking van onze RV32I processor. Deze is uitgewerkt vertrekkend van een (door ons) eerder geïmplementeerde Single Cycle processor. Door deze Single Cycle processor te pipelinen bekomen we een processor die efficiënter is. Deze RV32I processor is ontworpen in de programeer taal Verilog en getest in Modelsim. Hierna is deze via Quartus op een FPGA board gezet, hierdoor kan er ook GPIO's (General Purpose Input/Output) toegevoegd worden. In figuur 1 kan u het volledig schema van de RV32I processor terugvinden.

# 2 Instructies

Deze instructies zijn eerst toegevoegd in de eerder geimplemnteerde single cycle RISC-V Processor. Hier zijn alle instructies om te beurt getest voor een juiste werking te verifiëren. Hiervoor is de nodige assembler-code voor geschreven met behulp van VS CODE en RIPES. Hier kon met behulp van de testbench in ModelSim vergeleken worden wat op welk moment in het memory of registers kwam te zitten.

## 2.1 Lui and auipc groep

- `lui` (Load Upper Immediate): Zal een immediate waarde schrijven op het gegeven register.
- `auipc` (Add Upper Immediate Program Counter): Telt een upper immediate waarde op aan de PC en plaatst het resultaat in een register.

## 2.2 Jump groep

- `jal` (Jump And Link): de huidige PC waarde + 4 wordt opgeslagen in rd (register destination) en de PC zal verspringen naar het opgegeven adres (Immediate).
- `jalr` (Jump And Link Register): zal terug springen naar de PC die in het mee gegeven register staat en zal zijn huidige PC + 4 opslaan in een register.

## 2.3 Branch groep

- `beq` (Branch Equal): Zal springen naar het mee gegeven adres wanneer de 2 mee gegeven registers gelijk zijn aan elkaar.
- `bne` (Branch Not Equal): Zal springen naar het mee gegeven adres wanneer de 2 mee gegeven registers niet gelijk zijn aan elkaar.
- `blt` (Branch Less Then): Zal springen naar het mee gegeven adres wanneer het eerste register kleiner is dan het 2de register. Hier zal wel rekening gehouden met de signed bit (de meest significante bit).

- `bge` (Branch Greater Then): Zal springen naar het mee gegeven adres wanneer het eerste register groter is dan het 2de register. Hier zal wel rekening gehouden met de signed bit (de meest significante bit).
- `bltu` (Branch Less Then Unsigned): Doet hetzelfde als als 'blt' maar houdt geen rekening met de signed bit.
- `bgeu` (Branch Greater Then Unsigned): Doet hetzelfde als als 'bge' maar houdt geen rekening met de signed bit.

## 2.4 Load groep

- `lb` (Load Byte): Zal een waarde uit het Memory halen en deze opslaan in het mee gegeven register. Load Byte zal de 8 minst significante bits van het 32-bit woord opslaan. Deze instructie zal ook rekening houden met de signed bit van de waarde die uitgelezen wordt. Dit omdat deze bit verlengd zal moeten worden.
- `lbu` (Load Byte Unsigned): Zal hetzelfde doen als 'lb' maar houdt geen rekening met de singed bit van de waarde die uitgelezen wordt.
- `lh` (Load Halfword): Zal een waarde uit het Memory halen en deze opslaan in het mee gegeven register. Load Halfword zal de 16 minst significante bits van het 32-bit woord opslaan. Deze instructie zal ook rekening houden met de signed bit van de waarde die uitgelezen wordt. Dit omdat deze bit verlengd zal moeten worden.
- `lhu` (Load Halfword Unsigned): Zal hetzelfde doen als 'lh' maar houdt geen rekening met de singed bit van de waarde die uitgelezen wordt.
- `lw` (Load Word): Zal een waarde uit het Memory halen en deze opslaan in het mee gegeven register. Load Word zal het gehele 32-bit woord opslaan.

## 2.5 Store groep

- `sb` (Store Byte): Zal de waarde van het meegeven register opslaan in het Memory op het mee gegeven adres. Store Byte zal alleen de 7 minst significante bits van de 32-bit waarde opslaan.
- `sh` (Store Halfword): Zal de waarde van het meegeven register opslaan in het Memory op het mee gegeven adres. Store Halfword zal alleen de 16 minst significante bits van de 32-bit waarde opslaan.
- `sw` (Store Word): Zal de waarde van het meegeven register opslaan in het Memory op het mee gegeven adres. Store Word zal de volledige 32-bit waarde opslaan.

## 2.6 Arithmetic en logic R-type groep

Volgende Instructies behoren tot de arithmetic en logic R-Type groep. Bij R-type Instructies zullen er 2 registers gebruikt worden voor de operatie uit te voeren.

`add` (Add), `xor` (XOR), `or` (OR), `and` (AND): Hier worden logische operaties uitgevoerd met de waardes van de 2 mee gegeven registers. De uitkomst zal opgeslagen worden in het 'Destionation Register' (rd).

`slt` (Set less Than ), `sltu` (Set Less Than Unsigned): Zal een '1' in 'rd' zetten als de eerste waarde kleiner is dan de 2de. `slt` houdt wel rekening met de singed bit en `sltu` niet.

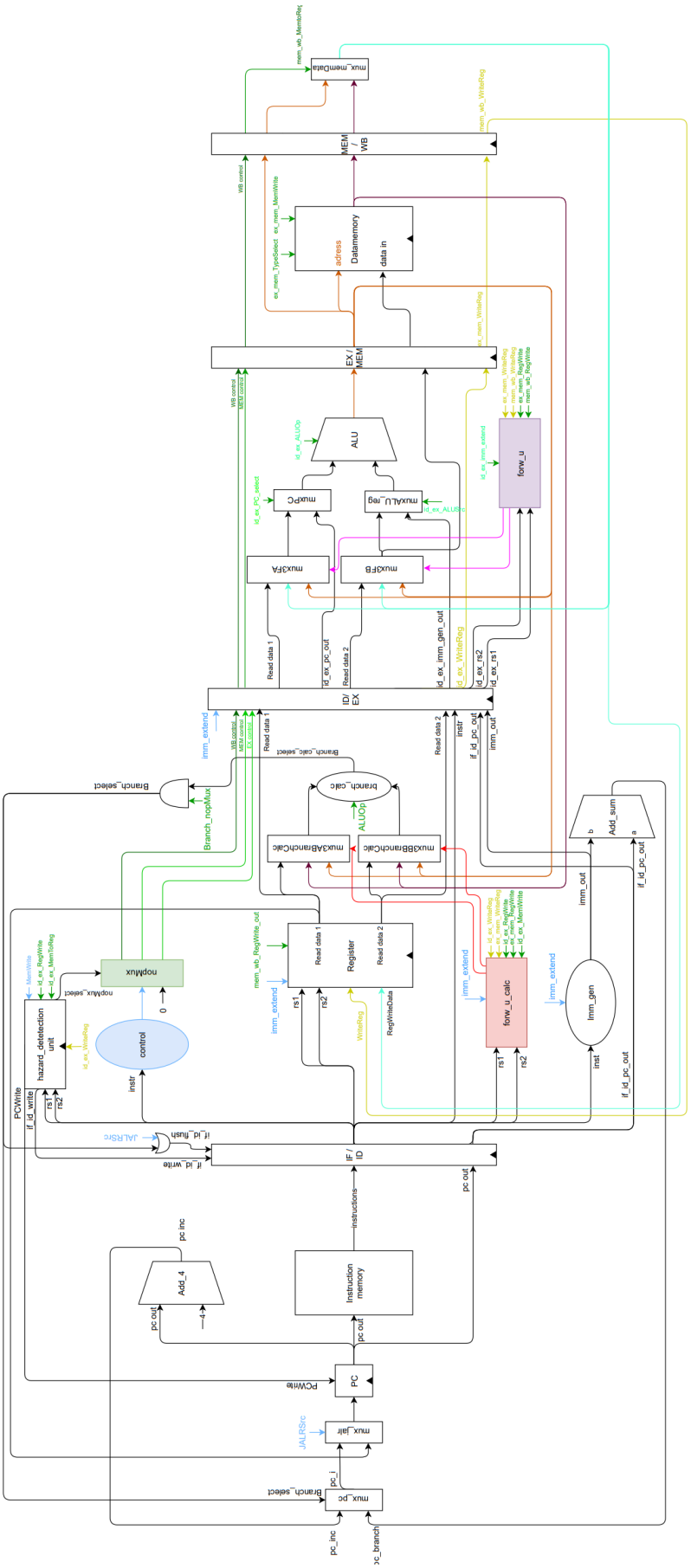
`sll` (Shift Left Logical ), `srl` (Shift Right Logical ), `sra` (Shift Right Arithmetic ): `sll` schuift de de bits een aantal keren op naar links. dit aantal wordt mee gegeven met de instructie. `srl` doet hetzelfde maar schuift de bits op naar recht.
`sra` schuift de bits ook op naar rechts met een mee geven aantal. Maar hier zal rekening gehouden worden met de signed bit. Deze bit zal dan links herhaald worden.

## 2.7 Arithmetic en logic immediate groep

Volgende Instructies behoren tot de arithmetic en logic immediate groep. Bij Immediate Instructies zal er 1 register gebruikt worden voor de operatie samen met een direct (immediate) toegevoegde waarde.

De logica van volgende instructies werden in 2.6 al aangehaald. Het verschil hier is dat de waarden die bij de R-type instructies uit het 2de register kwam, hier een directe (Immediate) waarde zal zijn.

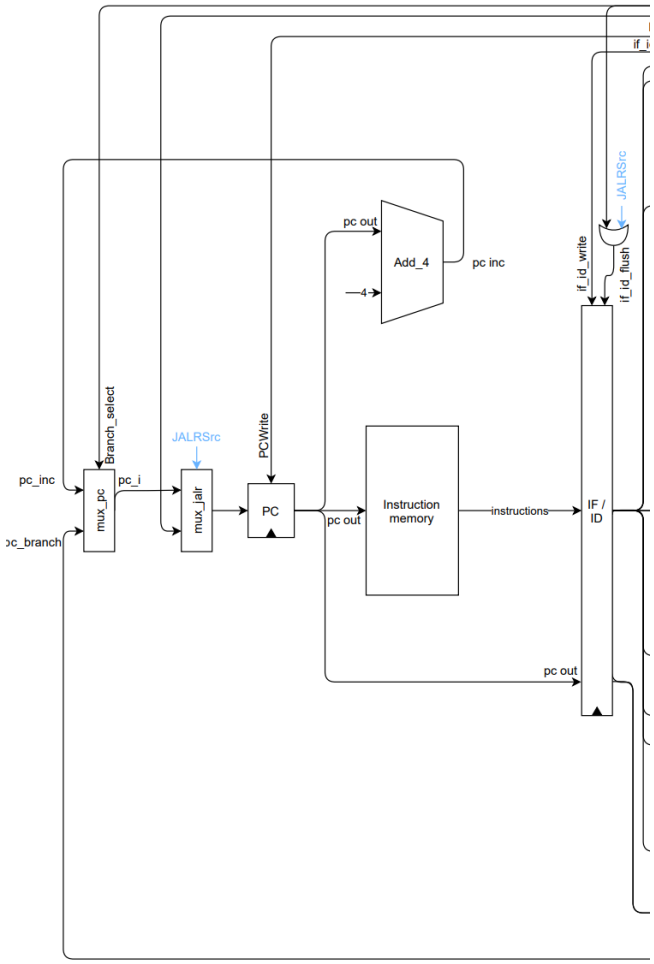
`addi` (Add Immediate), `slti` (Set less Than immediate), `sltiu` (Set Less Than Immediate Unsigned), `xori` (XOR Immediate), `ori` (OR Immediate), `andi` (AND Immediate), `slli` (Shift Left Logical Immediate), `srli` (Shift Right Logical Immediate), `srai` (Shift Right Arithmetic Immediate).




Figuur 1: Schema van de volledige RV32I processor

# 3 IF stage

Instruction Fetch (IF) is de eerste pipeline stage die aan bod komt. Hier zullen volgende modules gebruikt worden om de juiste instructies op te halen uit het Instruction Memory. In figuur 2 is deze stage te zien.




Figuur 2: Schematise weergaven van de Instroction Fetch (IF) stage

## 3.1 Program counter

De Program Counter is een geklokt register dat ervoor zal zorgen dat de counter elke clock-cycle doorgeven zal worden. Een van de input signalen van de PC is `PCWrite`. Dit signaal werkt als een enable signaal voor dit register en is afkomstig van de hazard detection unit die later vermeld wordt.

## 3.2 Mux program counter

Deze multiplexer zal met behulp van het controle signaal `Branch_Select` kiezen of de gewone program counter (PC + 4) zal doorgelaten worden of de program counter die nodig is als er gebrancht wordt. Dit komt omdat wanneer er gebrancht wordt, men meer dan 4 stappen verder zal moeten gaan. Het `Branch_Select` signaal zal hoog worden wanneer de instructie die uitgevoerd wordt daadwerkelijk een branch instructie is en wanneer de conditie van deze branch klopt.

## 3.3 Mux jalr

Deze multiplexer zal met behulp van het controle signaal `JALRSrc` kiezen of men de program counter van bovenstaande multiplexer moet nemen of de program counter dat uit return adress (register 1) komt. Wanneer men een Jal instructie uitvoert (een sprong naar een andere program counter), wordt de PC waar we naar moeten terugspringen opgeslagen in register 1. Als de instructie Jalr uitgevoerd wordt zal de controle blok (die later aan bod komt) het `JALRSrc` signaal hoog maken. Hierdoor krijgt de program counter de PC binnen waar we terug moeten naar springen.

## 3.4 Instruction Memory

De instruction memory zal met het `PC_OUT` signaal de juiste instructie ophalen en doorgeven naar zijn output. De instructies worden opgehaald uit het instruction memory. Deze worden ingeladen uit een .HEX bestand. Hier is dus gebruik gemaakt van de Verilog beschrijving.

## 3.5 Add 4

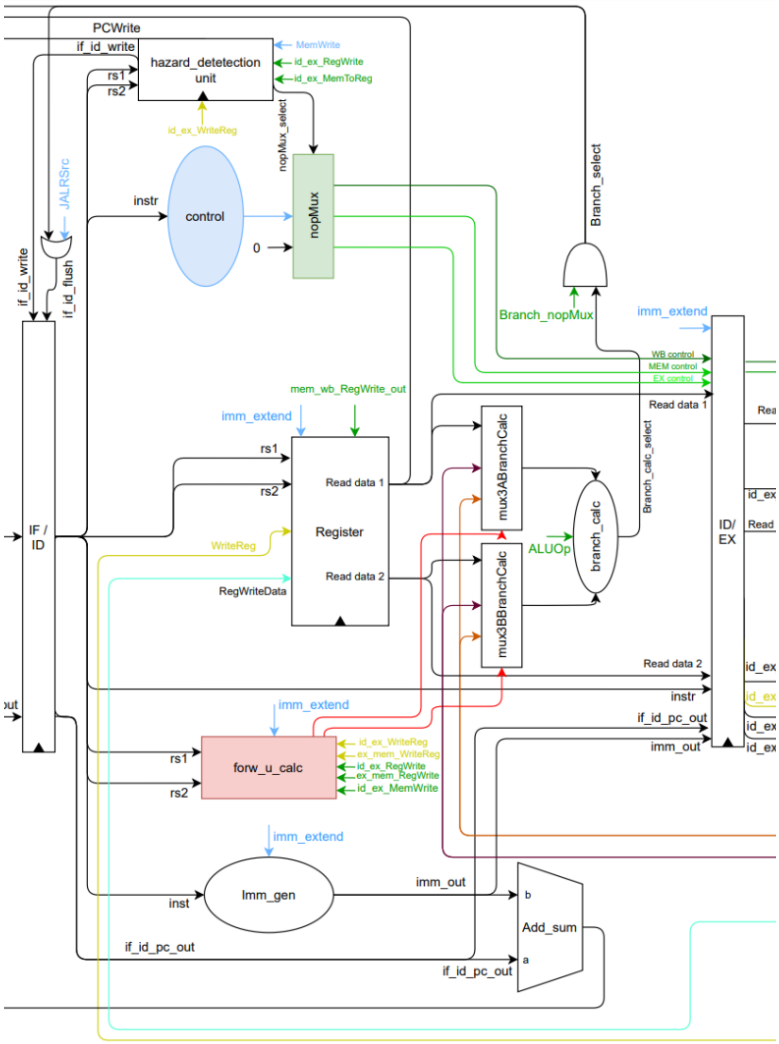
In deze module zal de binnekomende `PC_OUT` verhoogt worden met '4' en terug gestuurd worden naar de `Mux_program_counter` die hierboven al vermeld werd.

# 4 ID stage

Instruction Decode (ID) is de tweede pipeline stage die aan bod komt. Hier zullen volgende modules voornamelijk gebruikt worden om de reeds opgehaalde instructies te decoden en o.a. de nodige registers uit/in te lezen. In figuur 3 is deze stage te zien.

## 4.1 IF/ID register

De instructie die uit het Instruction memory komt, komt binnen in de IF-stage. Ook zal de program counter door dit register gestuurd worden. De hazard detection unit (waarover verder meer) zal wanneer hij zal stilvallen (stallen) het `if_id_write` signaal hoog maken. Wanneer er gebrancht wordt of de instructie `jalr` uitgevoerd wordt zal het if/id-register geflusht worden. Dit omdat er anders verkeerde data door de pipeline zal gaan. Deze komt van de instructie die zal uitgevoerd worden als de branch niet zou doorgaan.




Figuur 3: Schematische weergave van de Instruction Decode (ID) stage

## 4.2 Register

In het register worden alle 32 registers opgeslagen, deze waardes zijn uitkomsten van operaties tussen 2 registers en waardes die uit het RAM worden gelezen. Afhankelijk van de instructie weet het register welke van de 32 registers uitgelezen moet worden (dit kan er 1 of 2 zijn). Wanneer er simultaan op een register geschreven en gelezen wordt zal deze data direct door gekoppeld moeten worden, dit omdat er enkel geschreven wordt bij een stijgende flank van een clock-puls. Wanneer hier geen rekening mee zou gehouden worden, zal er op het moment van uitlezen dus nog een foute waarde staan.

Bij een immediate instructie zal de immediate deels op dezelfde plek staan in de 32 bit instructie, waar bij R, S en B instructies staat wat als 2de register gebruikt zal worden. Hierdoor bestaat de kans dat de waarde van de immediate overeen komt met het register waarop zal geschreven worden, wanneer hier geen rekening mee zou gehouden worden, wordt de data die geschreven wordt direct door gekoppeld naar read data 2.

| Instruction Type | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **R** register / register | funct7 | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | 1 | 1 |
| **I** immediates & loads | ± | imm [10: 0] | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | 1 | 1 |
| **S** stores | ± | imm [10: 5] | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm [4: 0] | | | | | opcode | | | | | 1 | 1 |
| **B** cond. branches | ± | imm [10: 5] | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm [4: 1] | | | | [11] | opcode | | | | | 1 | 1 |
| **U** upper immediates | ± | imm [30:12] | | | | | | | | | | | | | | | | | | | rd | | | | | opcode | | | | | 1 | 1 |
| **J** unconditional jumps | ± | imm [10: 1] | | | | | | | | | | [11] | imm [19:12] | | | | | | | | rd | | | | | opcode | | | | | 1 | 1 |

Figuur 4: Weergave 32-bit instructies types van de RV32I

## 4.3 Control

De Control module zal aan de hand van de binnenkomende instructie de controle signalen aansturen.

```
always @* begin
   Branch = 0; MemtoReg = 0; MemWrite = 0; ALUSrc = 0; RegWrite = 0;
   ALUOp = 0; Imm_extend = 0; PCSource = 0; jlrSrc = 0; Type_Select = 0;
   case (1'b1)
//-----lui and auipc group-----------------------------------------//
  lui_inst : begin Imm_extend = U_IMM; RegWrite = 1; ALUSrc = 1;
             ALUOp = ALU_ADD; end
  auipc_inst : begin Imm_extend = U_IMM; Branch = 1; PCSource =1;
               ALUOp = ALU_ADD; ALUSrc = 1; RegWrite = 1; end

//-----jump group---------------------------------------------------//
  jal_inst : begin Imm_extend = J_IMM; Branch =1; PCSource =1;
             ALUOp = ALU_JAL; RegWrite = 1; end
  jlr_inst: begin Imm_extend = I_IMM; jlrSrc = 1; PCSource =1;
             ALUOp = ALU_JAL; RegWrite = 1; end

//-----branch group-------------------------------------------------//
  beq_inst  : begin ALUOp = ALU_SUB; Branch = 1; Imm_extend = B_IMM; end
  bne_inst  : begin Imm_extend = B_IMM; ALUOp = ALU_BRANCH; Branch = 1; end
  blt_inst  : begin Imm_extend = B_IMM; ALUOp = ALU_BLT; Branch = 1; end
  bge_inst  : begin Imm_extend = B_IMM; ALUOp = ALU_BGE; Branch = 1; end
  bltu_inst   : begin Imm_extend = B_IMM; ALUOp = ALU_BLTU; Branch = 1; end
  bgeu_inst   : begin Imm_extend = B_IMM; ALUOp = ALU_BGEU; Branch = 1; end
```

```
24
25 //-----load group-----------------------------------------------------//
26   lb_inst    : begin Imm_extend = I_IMM; ALUOp = ALU_ADD; ALUSrc = 1;
27                  MemtoReg = 1; RegWrite = 1; Type_Select = BYTE; end
28   lbu_inst   : begin Imm_extend = I_IMM; ALUOp = ALU_ADD; ALUSrc = 1;
29                  MemtoReg = 1; RegWrite = 1; Type_Select = BYTEU; end
30   lh_inst    : begin Imm_extend = I_IMM; ALUOp = ALU_ADD; ALUSrc = 1;
31                  MemtoReg = 1; RegWrite = 1; Type_Select = HALF_WORD; end
32   lhu_inst   : begin Imm_extend = I_IMM; ALUOp = ALU_ADD; ALUSrc = 1;
33                  MemtoReg = 1; RegWrite = 1; Type_Select = HALF_WORDU; end
34   lw_inst    : begin Imm_extend = I_IMM; ALUOp = ALU_ADD; ALUSrc = 1;
35                  MemtoReg = 1; RegWrite = 1; Type_Select = WORD; end
36
37 //-----store group----------------------------------------------------//
38     sb_inst    : begin Imm_extend = S_IMM; ALUOp = ALU_ADD; ALUSrc = 1;
39                    MemWrite = 1; Type_Select = BYTE; end
40     sh_inst    : begin Imm_extend = S_IMM; ALUOp = ALU_ADD; ALUSrc = 1;
41                    MemWrite = 1; Type_Select = HALF_WORD; end
42   sw_inst    : begin Imm_extend = S_IMM; ALUOp = ALU_ADD; ALUSrc = 1;
43                  MemWrite = 1; Type_Select = WORD; end
44
45 //-----arithmetic & logic immediate group-----------------------------//
46   addi_inst    : begin Imm_extend = I_IMM; RegWrite = 1; ALUSrc = 1;
47                  ALUOp = ALU_ADD; end
48   slti_inst    : begin Imm_extend = I_IMM; RegWrite = 1; ALUSrc = 1;
49                  ALUOp = ALU_SLT; end
50     sltiu_inst   : begin Imm_extend = I_IMM; RegWrite = 1; ALUSrc = 1;
51                    ALUOp = ALU_SLTU; end
52   xori_inst    : begin Imm_extend = I_IMM; RegWrite = 1; ALUSrc = 1;
53                  ALUOp = ALU_XOR; end
54   ori_inst   : begin Imm_extend = I_IMM; RegWrite = 1; ALUSrc = 1;
55                  ALUOp = ALU_OR; end
56   andi_inst    : begin Imm_extend = I_IMM; RegWrite = 1; ALUSrc = 1;
57                  ALUOp = ALU_AND; end
58   slli_inst    : begin Imm_extend = I_IMM; RegWrite = 1; ALUSrc = 1;
59                  ALUOp = ALU_SLL; end
60   srli_inst    : begin Imm_extend = I_IMM; RegWrite = 1; ALUSrc = 1;
61                  ALUOp = ALU_SRL; end
62   srai_inst    : begin Imm_extend = I_IMM; RegWrite = 1; ALUSrc = 1;
63                  ALUOp = ALU_SRA; end
64
65 //-----arithmetic & logic R-type instructions-------------------------//
66   add_inst   : begin ALUOp = ALU_ADD; RegWrite = 1; end
67   sub_inst   : begin ALUOp = ALU_SUB; RegWrite = 1; end
68   sll_inst   : begin ALUOp = ALU_SLL; RegWrite = 1; end
69   slt_inst   : begin ALUOp = ALU_SLT; RegWrite = 1; end
70   sltu_inst    : begin ALUOp = ALU_SLTU; RegWrite = 1; end
71   xor_inst   : begin ALUOp = ALU_XOR; RegWrite = 1; end
72   srl_inst   : begin ALUOp = ALU_SRL; RegWrite = 1; end
73     sra_inst   : begin ALUOp = ALU_SRA; RegWrite = 1; end
74   or_inst    : begin ALUOp = ALU_OR; RegWrite = 1; end
75   and_inst   : begin ALUOp = ALU_AND; RegWrite = 1; end
76
77   default: ; // Here we need to implement an exception (currently NOP)
78     endcase
79 end
```

Listing 1: Control case in Verilog

## 4.4 Immediate generator

Zoals te zien in figuur 4 heeft elk instructie type (buiten R-type) een immediate waarde. Deze staan bij elke type instructie op een andere plek in de binnenkomende 32-bit instructie. In de Immediate generator worden deze opgedeelde immediates vanuit de instructie aan elkaar "geplakt"tot een geheel 32-bit getal. Dit wordt voor elk type op een andere manier gedaan.

## 4.5 Add sum

In deze adder wordt de program counter opgeteld samen met de waarde die uit de immediate generator komt. Deze waarde wordt op zijn beurt teruggekoppeld naar de mux programcounter die hierboven al besproken werd.

## 4.6 Branch Calculator

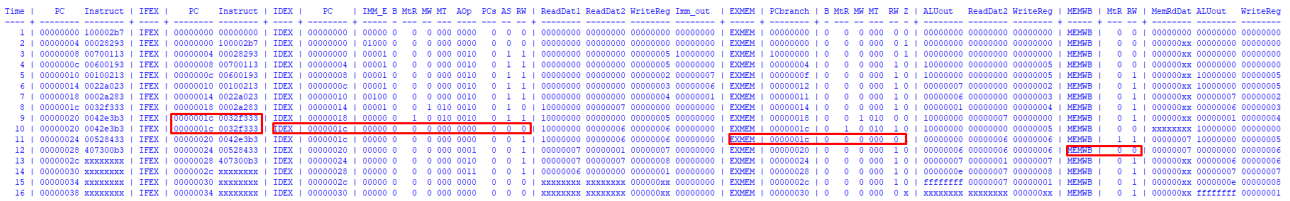
In de branch calculater wordt er gekeken of de conditie van de branch klopt, wanneer deze klopt zal het `branch_calc_select` signaal hoog worden. Afhankelijk van het `ALUOp` signaal dat van de nopMux komt zal de calculater weten welke operatie hij moet uitvoeren. Deze module wordt in deze stage van de pipeline uitgevoerd zodat de Delay of Branches zo klein mogelijk is (er moet maar 1 maal geflusht worden in het IF/ID register).

## 4.7 Forwarding Unit / mux

Essentieel werd er alleen gebruikt gemaakt van één enkele forwarding unit die in de EX-stage stond. Na het testen van de instructies in de pipeline-versie bleek dit niet genoeg te zijn. Hierna werd besloten een tweede forwarding unit te maken in de ID-stage. Deze forwarding Unit werk essentieel op dezelfde manier als in sectie 5.2 zal uitgelegd worden. Enkel wordt hier extra rekening gehouden met wanneer er een Store instructie voor een Branch instructie zal gebeuren. Dit omdat de branch anders de verkeerde waarde uit de 3-poort-mux (mux3BBranchCalc) zou krijgen, hiervoor hebben we een extra signaal `id_ex_MemWrite` moeten toevoegen.

## 4.8 Hazard detection unit / nopMux

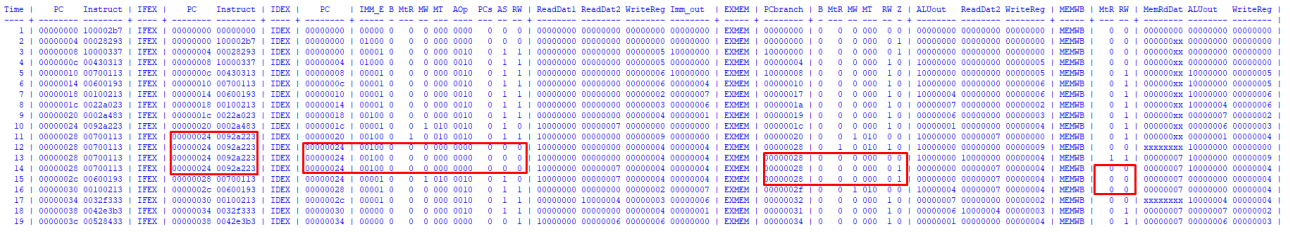
Wanneer er een waarde, door een load-instructie uit te voeren, in een register wordt opgeslagen en er in een volgende instructie van dit register gebruik gemaakt wordt, dan zal de hele pipeline moeten gestald worden. De data hazard detection zal dit opmerken en een nop (No Operation) uitvoeren. Dit door het IF/ID register 1 clock-cycli te stallen (de `enable` op '0' zetten) en alle controle signalen voor 1 periode op 0 te zetten (`nopMux_select` = 1). Deze module is getest door de opeen volgende instructies in figuur 4.57 op pagina 305 in the PH book RISC-V edition. Hier wordt een and-instructie na een load-instructie gedaan, hier zal dan 1 nop gemaakt worden. In foto 5 wordt een and-instructie na een load-instrcutie uitgevoerd. In de Testbench is te zien dat hier effectief een 'nop' wordt uitgevoerd in de rode kaders.



| Time | PC | Instruct | IFEX | PC | Instruct | IDEX | PC | IMM_E | B | MtR | MW | MT | AOp | PCs | AS | RW | ReadDat1 | ReadDat2 | WriteReg | Imm_out | EXMEM | PCbranch | B | MtR | MW | MT | RW | Z | ALUout | ReadDat2 | WriteReg | MEMWB | MtR | RW | MemRdDat | ALUout | WriteReg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 00000000 | 100002b7 | IFEX | 00000000 | 00000000 | IDEX | 00000000 | 00000 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 0 | 00000000 | 00000000 | 00000000 | 00000000 | EXMEM | 00000000 | 0 | 0 | 0 | 000 | 0 | 0 | 00000000 | 00000000 | 00000000 | MEMWB | 0 | 0 | 00000000 | 00000000 | 00000000 |
| 2 | 00000004 | 00028293 | IFEX | 00000000 | 100002b7 | IDEX | 00000000 | 01000 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 0 | 00000000 | 00000000 | 00000000 | 00000000 | EXMEM | 00000000 | 0 | 0 | 0 | 000 | 0 | 1 | 00000000 | 00000000 | 00000000 | MEMWB | 0 | 0 | 000000xx | 00000000 | 00000000 |
| 3 | 00000008 | 00700113 | IFEX | 00000004 | 00028293 | IDEX | 00000000 | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000005 | 10000000 | EXMEM | 10000000 | 0 | 0 | 0 | 000 | 0 | 1 | 00000000 | 00000000 | 00000000 | MEMWB | 0 | 0 | 000000xx | 00000000 | 00000000 |
| 4 | 0000000c | 00600193 | IFEX | 00000008 | 00700113 | IDEX | 00000004 | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000005 | 00000005 | EXMEM | 00000004 | 0 | 0 | 0 | 000 | 1 | 0 | 10000000 | 00000000 | 00000005 | MEMWB | 0 | 0 | 000000xx | 00000000 | 00000000 |
| 5 | 00000010 | 00100213 | IFEX | 0000000c | 00600193 | IDEX | 00000008 | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000002 | 00000007 | EXMEM | 0000000f | 0 | 0 | 0 | 000 | 1 | 0 | 10000000 | 00000000 | 00000005 | MEMWB | 0 | 1 | 000000xx | 10000000 | 00000005 |
| 6 | 00000014 | 0022a023 | IFEX | 00000010 | 00100213 | IDEX | 0000000c | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000003 | 00000006 | EXMEM | 00000012 | 0 | 0 | 0 | 000 | 1 | 0 | 00000007 | 00000000 | 00000002 | MEMWB | 0 | 1 | 000000xx | 10000000 | 00000005 |
| 7 | 00000018 | 0002a283 | IFEX | 00000014 | 0022a023 | IDEX | 00000010 | 00100 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000004 | 00000001 | EXMEM | 00000011 | 0 | 0 | 0 | 000 | 1 | 0 | 00000006 | 00000000 | 00000003 | MEMWB | 0 | 1 | 000000xx | 00000007 | 00000002 |
| 8 | 0000001c | 0032f333 | IFEX | 00000018 | 0002a283 | IDEX | 00000014 | 00001 | 0 | 1 | 0 | 010 | 0010 | 0 | 1 | 0 | 10000000 | 00000007 | 00000000 | 00000000 | EXMEM | 00000014 | 0 | 0 | 0 | 000 | 1 | 0 | 00000001 | 00000000 | 00000004 | MEMWB | 0 | 1 | 000000xx | 00000006 | 00000003 |
| 9 | 00000020 | 0042e3b3 | IFEX | 0000001c | 0032f333 | IDEX | 00000018 | 00000 | 0 | 1 | 0 | 010 | 0010 | 0 | 1 | 1 | 10000000 | 00000000 | 00000005 | 00000000 | EXMEM | 00000018 | 0 | 0 | 1 | 010 | 0 | 0 | 10000000 | 00000007 | 00000000 | MEMWB | 0 | 1 | 000000xx | 00000001 | 00000004 |
| 10 | 00000020 | 0042e3b3 | IFEX | 0000001c | 0032f333 | IDEX | 0000001c | 00000 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 0 | 10000000 | 00000006 | 00000006 | 00000000 | EXMEM | 0000001c | 0 | 1 | 0 | 010 | 1 | 0 | 10000000 | 00000000 | 00000005 | MEMWB | 0 | 0 | xxxxxxxx | 10000000 | 00000000 |
| 11 | 00000024 | 00528433 | IFEX | 00000020 | 0042e3b3 | IDEX | 0000001c | 00000 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 1 | 10000000 | 00000006 | 00000006 | 00000000 | EXMEM | 0000001c | 0 | 0 | 0 | 000 | 0 | 0 | 00000000 | 00000006 | 00000006 | MEMWB | 1 | 1 | 00000007 | 10000000 | 00000005 |
| 12 | 00000028 | 407300b3 | IFEX | 00000024 | 00528433 | IDEX | 00000020 | 00000 | 0 | 0 | 0 | 000 | 0001 | 0 | 0 | 1 | 00000007 | 00000001 | 00000007 | 00000000 | EXMEM | 00000020 | 0 | 0 | 0 | 000 | 1 | 0 | 00000006 | 00000006 | 00000006 | MEMWB | 0 | 0 | 00000007 | 00000000 | 00000006 |
| 13 | 0000002c | xxxxxxxx | IFEX | 00000028 | 407300b3 | IDEX | 00000024 | 00000 | 0 | 0 | 0 | 000 | 0010 | 0 | 0 | 1 | 00000007 | 00000007 | 00000008 | 00000000 | EXMEM | 00000024 | 0 | 0 | 0 | 000 | 1 | 0 | 00000007 | 00000001 | 00000007 | MEMWB | 0 | 1 | 000000xx | 00000006 | 00000006 |
| 14 | 00000030 | xxxxxxxx | IFEX | 0000002c | xxxxxxxx | IDEX | 00000028 | 00000 | 0 | 0 | 0 | 000 | 0011 | 0 | 0 | 1 | 00000006 | 00000000 | 00000001 | 00000000 | EXMEM | 00000028 | 0 | 0 | 0 | 000 | 1 | 0 | 0000000e | 00000007 | 00000008 | MEMWB | 0 | 1 | 000000xx | 00000007 | 00000007 |
| 15 | 00000034 | xxxxxxxx | IFEX | 00000030 | xxxxxxxx | IDEX | 0000002c | 00000 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 0 | xxxxxxxx | xxxxxxxx | 000000xx | 00000000 | EXMEM | 0000002c | 0 | 0 | 0 | 000 | 1 | 0 | ffffffff | 00000007 | 00000001 | MEMWB | 0 | 1 | 000000xx | 0000000e | 00000008 |
| 16 | 00000038 | xxxxxxxx | IFEX | 00000034 | xxxxxxxx | IDEX | 00000030 | 00000 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 0 | xxxxxxxx | xxxxxxxx | 000000xx | 00000000 | EXMEM | 00000030 | 0 | 0 | 0 | 000 | 0 | x | xxxxxxxx | xxxxxxxx | 000000xx | MEMWB | 0 | 1 | 000000xx | ffffffff | 00000001 |

Figuur 5: Testbench in ModelSim simulatie van 1 No Opereation

Na het testen van andere instructies die na een load-instructie plaats vinden is het volgende ondervonden. Wanneer er na deze load-instructie een store-instructie wordt uitgevoerd op eenzelfde register werkt vorige implementatie niet. Er zal 3x gestald moeten worden, omdat de waarde van de load-instructie al in het register moet staan voordat de store-instructie in de ID stage terecht komt. De werking hiervan is te zien in foto 6 waar in de rode kaders 3 nop's te zien zijn.

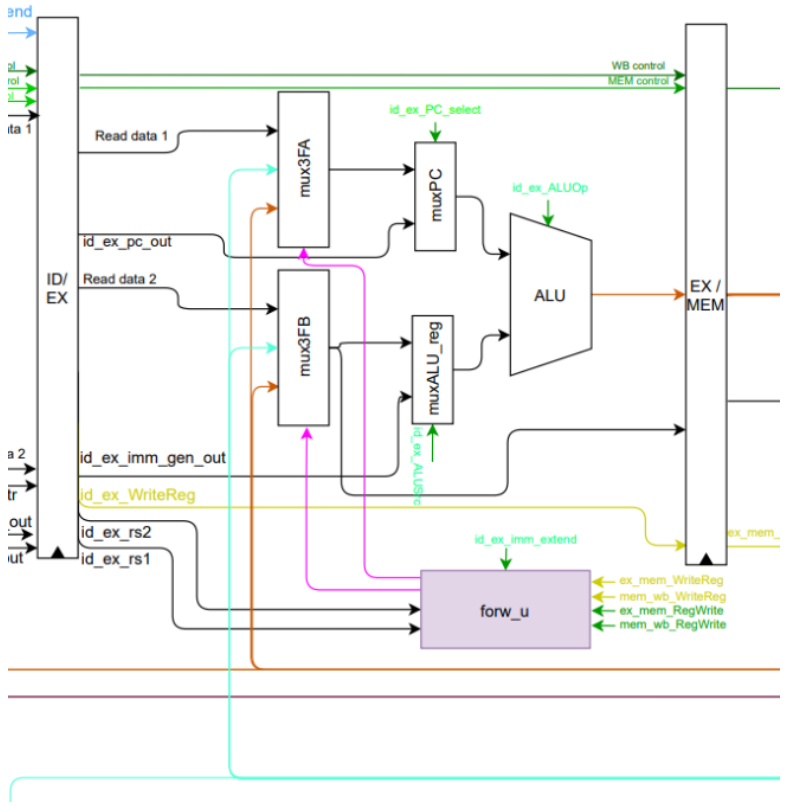


| Time | PC | Instruct | IFEX | PC | Instruct | IDEX | PC | IMM_E | B | MtR | MW | MT | AOp | PCs | AS | RW | ReadDat1 | ReadDat2 | WriteReg | Imm_out | EXMEM | PCbranch | B | MtR | MW | MT | RW | Z | ALUout | ReadDat2 | WriteReg | MEMWB | MtR | RW | MemRdDat | ALUout | WriteReg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 00000000 | 100002b7 | IFEX | 00000000 | 00000000 | IDEX | 00000000 | 00000 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 0 | 00000000 | 00000000 | 00000000 | 00000000 | EXMEM | 00000000 | 0 | 0 | 0 | 000 | 0 | 0 | 00000000 | 00000000 | 00000000 | MEMWB | 0 | 0 | 00000000 | 00000000 | 00000000 |
| 2 | 00000004 | 00028293 | IFEX | 00000000 | 100002b7 | IDEX | 00000000 | 01000 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 0 | 00000000 | 00000000 | 00000000 | 00000000 | EXMEM | 00000000 | 0 | 0 | 0 | 000 | 0 | 1 | 00000000 | 00000000 | 00000000 | MEMWB | 0 | 0 | 000000xx | 00000000 | 00000000 |
| 3 | 00000008 | 10000337 | IFEX | 00000004 | 00028293 | IDEX | 00000000 | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000005 | 10000000 | EXMEM | 10000000 | 0 | 0 | 0 | 000 | 0 | 1 | 00000000 | 00000000 | 00000000 | MEMWB | 0 | 0 | 000000xx | 00000000 | 00000000 |
| 4 | 0000000c | 00430313 | IFEX | 00000008 | 10000337 | IDEX | 00000004 | 01000 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000005 | 00000005 | EXMEM | 00000004 | 0 | 0 | 0 | 000 | 1 | 0 | 10000000 | 00000000 | 00000005 | MEMWB | 0 | 0 | 000000xx | 00000000 | 00000000 |
| 5 | 00000010 | 00700113 | IFEX | 0000000c | 00430313 | IDEX | 00000008 | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000006 | 10000000 | EXMEM | 10000008 | 0 | 0 | 0 | 000 | 1 | 0 | 10000000 | 00000000 | 00000005 | MEMWB | 0 | 1 | 000000xx | 10000000 | 00000005 |
| 6 | 00000014 | 00600193 | IFEX | 00000010 | 00700113 | IDEX | 0000000c | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000006 | 00000004 | EXMEM | 00000010 | 0 | 0 | 0 | 000 | 1 | 0 | 10000000 | 00000000 | 00000006 | MEMWB | 0 | 1 | 000000xx | 10000000 | 00000005 |
| 7 | 00000018 | 00100213 | IFEX | 00000014 | 00600193 | IDEX | 00000010 | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000002 | 00000007 | EXMEM | 00000017 | 0 | 0 | 0 | 000 | 1 | 0 | 10000004 | 00000000 | 00000006 | MEMWB | 0 | 1 | 000000xx | 10000000 | 00000006 |
| 8 | 0000001c | 0022a023 | IFEX | 00000018 | 00100213 | IDEX | 00000014 | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000003 | 00000006 | EXMEM | 0000001a | 0 | 0 | 0 | 000 | 1 | 0 | 00000007 | 00000000 | 00000002 | MEMWB | 0 | 1 | 000000xx | 10000004 | 00000006 |
| 9 | 00000020 | 0002a483 | IFEX | 0000001c | 0022a023 | IDEX | 00000018 | 00100 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000004 | 00000001 | EXMEM | 00000019 | 0 | 0 | 0 | 000 | 1 | 0 | 00000006 | 00000000 | 00000003 | MEMWB | 0 | 1 | 000000xx | 00000007 | 00000002 |
| 10 | 00000024 | 0092a223 | IFEX | 00000020 | 0002a483 | IDEX | 0000001c | 00001 | 0 | 0 | 1 | 010 | 0010 | 0 | 1 | 0 | 10000000 | 00000007 | 00000000 | 00000000 | EXMEM | 0000001c | 0 | 0 | 0 | 000 | 1 | 0 | 00000001 | 00000000 | 00000004 | MEMWB | 0 | 1 | 000000xx | 00000006 | 00000003 |
| 11 | 00000028 | 00700113 | IFEX | 00000024 | 0092a223 | IDEX | 00000020 | 00100 | 0 | 1 | 0 | 010 | 0010 | 0 | 1 | 1 | 10000000 | 00000000 | 00000009 | 00000000 | EXMEM | 00000020 | 0 | 0 | 1 | 010 | 0 | 0 | 10000000 | 00000007 | 00000000 | MEMWB | 0 | 1 | 000000xx | 00000001 | 00000004 |
| 12 | 00000028 | 00700113 | IFEX | 00000024 | 0092a223 | IDEX | 00000024 | 00100 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 0 | 10000000 | 00000000 | 00000004 | 00000004 | EXMEM | 00000028 | 0 | 1 | 0 | 010 | 1 | 0 | 10000000 | 00000000 | 00000009 | MEMWB | 0 | 0 | xxxxxxxx | 10000000 | 00000000 |
| 13 | 00000028 | 00700113 | IFEX | 00000024 | 0092a223 | IDEX | 00000024 | 00100 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 0 | 10000000 | 00000000 | 00000004 | 00000004 | EXMEM | 00000028 | 0 | 0 | 0 | 000 | 0 | 0 | 10000000 | 10000000 | 00000004 | MEMWB | 1 | 1 | 00000007 | 10000000 | 00000009 |
| 14 | 00000028 | 00700113 | IFEX | 00000024 | 0092a223 | IDEX | 00000024 | 00100 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 0 | 10000000 | 00000007 | 00000004 | 00000004 | EXMEM | 00000028 | 0 | 0 | 0 | 000 | 0 | 1 | 00000000 | 00000007 | 00000004 | MEMWB | 0 | 0 | 00000007 | 10000000 | 00000004 |
| 15 | 0000002c | 00600193 | IFEX | 00000028 | 00700113 | IDEX | 00000024 | 00001 | 0 | 0 | 1 | 010 | 0010 | 0 | 1 | 0 | 10000000 | 00000007 | 00000004 | 00000004 | EXMEM | 00000028 | 0 | 0 | 0 | 000 | 0 | 1 | 00000000 | 00000007 | 00000004 | MEMWB | 0 | 0 | 00000007 | 00000000 | 00000004 |
| 16 | 00000030 | 00100213 | IFEX | 0000002c | 00600193 | IDEX | 00000028 | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000002 | 00000007 | EXMEM | 0000002f | 0 | 0 | 1 | 010 | 0 | 0 | 10000004 | 00000007 | 00000004 | MEMWB | 0 | 0 | 00000007 | 00000000 | 00000004 |
| 17 | 00000034 | 0032f333 | IFEX | 00000030 | 00100213 | IDEX | 0000002c | 00001 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 10000004 | 00000003 | 00000006 | EXMEM | 00000032 | 0 | 0 | 0 | 000 | 1 | 0 | 00000007 | 00000000 | 00000002 | MEMWB | 0 | 0 | xxxxxxxx | 10000004 | 00000004 |
| 18 | 00000038 | 0042e3b3 | IFEX | 00000034 | 0032f333 | IDEX | 00000030 | 00000 | 0 | 0 | 0 | 000 | 0010 | 0 | 1 | 1 | 00000000 | 00000000 | 00000004 | 00000001 | EXMEM | 00000031 | 0 | 0 | 0 | 000 | 1 | 0 | 00000006 | 10000004 | 00000003 | MEMWB | 0 | 1 | 00000007 | 00000007 | 00000002 |
| 19 | 0000003c | 00528433 | IFEX | 00000038 | 0042e3b3 | IDEX | 00000034 | 00000 | 0 | 0 | 0 | 000 | 0000 | 0 | 0 | 1 | 10000000 | 00000006 | 00000006 | 00000000 | EXMEM | 00000034 | 0 | 0 | 0 | 000 | 1 | 0 | 00000001 | 00000000 | 00000004 | MEMWB | 0 | 1 | 00000007 | 00000006 | 00000003 |

Figuur 6: Testbench in ModelSim simulatie van 3 No Opereations

# 5 EX stage

Execute (EX) is de derde pipeline stage die aan bod komt. Hier zullen volgende modules voornamelijk gebruikt worden om met de reeds uitgelezen registers operaties uit te voeren. In figuur 7 is deze stage te zien.




Figuur 7: Schematise weergaven van de Execute (EX) stage

## 5.1 ID/EX register

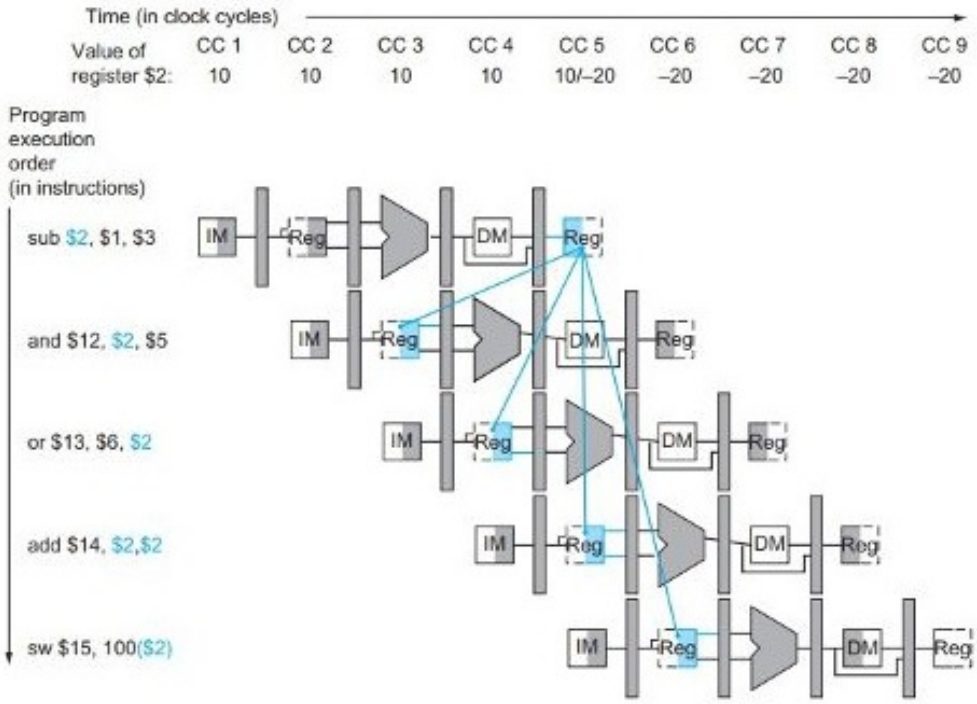
In dit register zullen alle controle signalen, die we in de volgende stages nodig hebben, doorverbonden worden. De 2 uitgelezen waardes van de registers zullen ook doorverbonden worden, samen met program counter, de output van de immediate generator en de nodige delen van de instructie.

## 5.2 Forwarding unit / mux

Zoals in onderstaande foto te zien is (figuur 8), ontstaat er een probleem wanneer er in opeenvolgende instructies geschreven en daarna gelezen wordt op eenzelfde register. Wanneer er hier geen forwarding gebruikt zou worden zou de pipeline meerdere clock pulsen gestald moeten worden om correct te kunnen werken. Dit probleem kan worden opgelost door terugkoppeling van waardes uit volgende stages (vorige instructies) samen met onderstaande case (die in de forwarding unit voorkomt). Deze case bepaalt het select-signaal voor de 2 3-poorts multiplexers die te zien zijn op figuur 3, hierdoor zal de pipeline niet meer gestald moeten worden en zal hierdoor dus geen tijd verliezen.

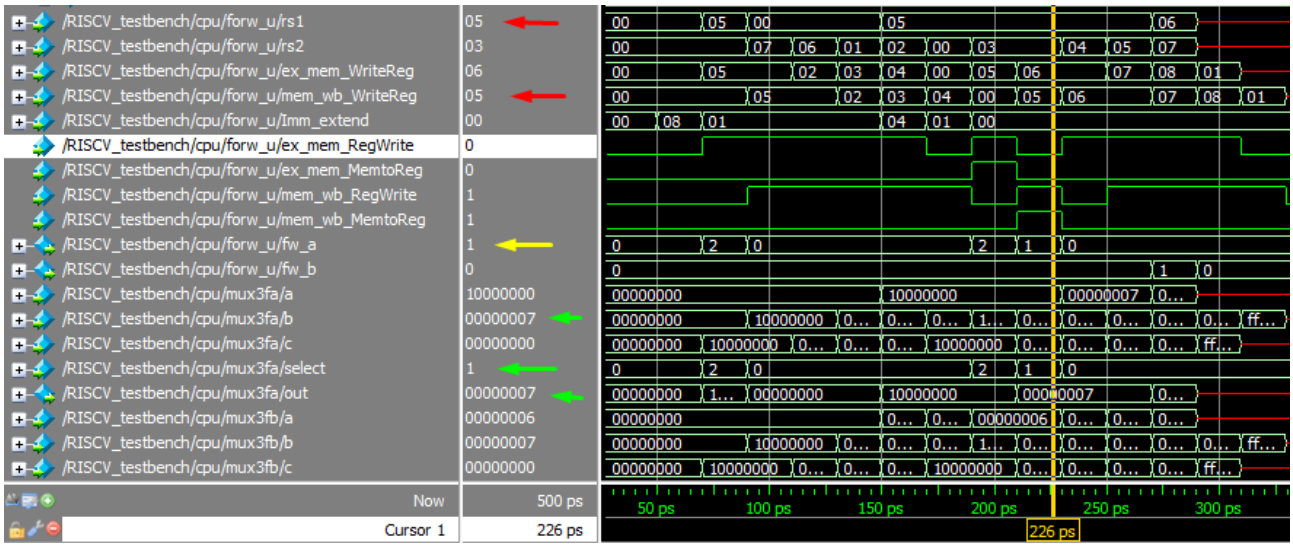
```
always @(*) begin

  fw_a = 0; fw_b = 0;

   if ((mem_wb_RegWrite)&&(mem_wb_WriteReg != 0)&&(mem_wb_WriteReg == rs1))
        begin
            if ((ex_mem_RegWrite) && (ex_mem_WriteReg != 0) &&
            (ex_mem_WriteReg == rs1)) begin end
            else begin fw_a <= 2'b01; end
        end
   if ((mem_wb_RegWrite) && (mem_wb_WriteReg != 0) &&
      (mem_wb_WriteReg == rs2) && (Imm_extend != 5'b00001))
        begin
            if ((ex_mem_RegWrite) && (ex_mem_WriteReg != 0) &&
               (ex_mem_WriteReg == rs2)) begin end
            else begin fw_b <= 2'b01; end
        end
   if ((ex_mem_RegWrite) && (ex_mem_WriteReg != 0) &&
      (ex_mem_WriteReg == rs1)) begin fw_a <= 2'b10; end
   if ((ex_mem_RegWrite) && (ex_mem_WriteReg != 0) &&
      (ex_mem_WriteReg == rs2) && (Imm_extend != 5'b00001))
      begin fw_b <= 2'b10; end
end
```

Listing 2: Forwarding unit in Verilog




Figuur 8: Verduidelijking nut pipeline

De implementatie van de forwarding unit is getest door middel van het bekijken van de wave simulatie in Modelsim. In figuur 9 is te zien hoe er nagekeken werd of de implementatie klopte. Bij de rode pijltjes is te zien dat er op het moment dat er in register x5 gelezen wordt, er 2 instructies eerder op dit register al geschreven werd. Hierdoor zal het selectie-signaal naar de mux (mux3fa) een waarde van 1 krijgen en zal er aan forwarding gedaan worden. Deze methode hebben we voor alle mogelijke forwarding problemen nagekenen.




Figuur 9: Wave simulatie van het gebruik van de forwarding Unit

## 5.3 MuxALU reg / muxPC

De multiplexer ALU reg zal met behulp van het signaal `id_ex_ALUSrc` bepalen of de uitgelezen waarde van de 3-poorts mux (mux3FB) of de output van de immediate generator gebruikt zal worden om de operaties mee uit te voeren.

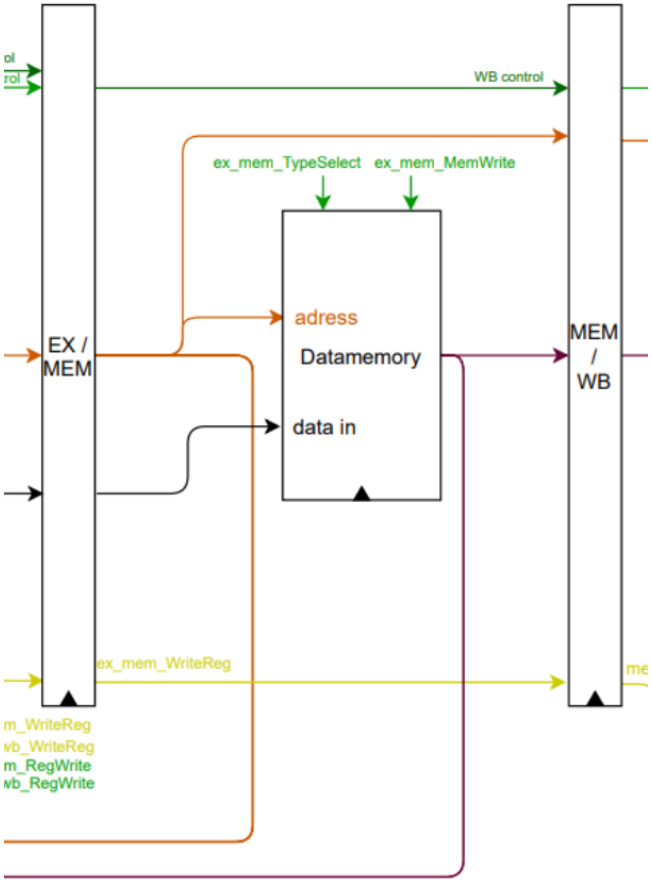
De multiplexer muxPC zal met behulp van het signaal `id_ex_PC_select` bepalen of de uitgelezen waarde de 3-poorts mux (mux3FA) of de program counter zal gebruikt worden om de operaties mee uit te voeren. Het selectiesignaal `id_ex_PC_select` zal enkel bij de instructies auipc en jlr hoog worden.

## 5.4 ALU

De Arithmetic logic unit (ALU) zal afhankelijk van het signaal `id_ed_ALUOp` bepalen welke operatie die zal uitvoeren met de twee binnenkomende waardes, dit kan een and, or, add, subtract, xor, ... zijn.

# 6 Mem stage

Memory acces (MEM) is het vierde pipeline stage die aan bod komt. In deze stage kan er geschreven worden naar het memory of kan er uit gelezen worden . In figuur 10 is deze stage te zien.




Figuur 10: Schematise weergaven van de Memory acces (MEM) stage

## 6.1 EX/MEM register

Net zoals in het ID/EX register zal het EX/MEM register de nodige controle signalen voor de volgende stage doorkoppelen, ook zal hier de uitkomst van de ALU, de uitkomst van de mux3FB multiplier en de waarde van het writeback registeer doorgekoppeld worden.

## 6.2 Datamemory

Wanneer het controlesignaal `ex_mem_MemWrite` hoog is zal er in het datamemory geschreven kunnen worden. Afhankelijk van het schrijfadres kan er met een case gewerkt worden om de juiste deel in het memory te bepalen (Memory, stack of GPIO). Het memory begint bij adres 10000, de stack met 7ffff en de GPIO's bij 40000. Dit is te zien in Listing 3.

Bij de instructies van de load en de store groep zal er rekening gehouden moeten worden of dit een word, half word of byte is. Ook zal er rekening gehouden moeten worden of dit een signed of unsigned instructie is. Dit hebben we opgelost door een extra signaal `ex_mem_TypeSelect`. Hierdoor weten we of we een deel van het adres moeten uitlezen en of we rekening moeten houden met de meest significante bit.

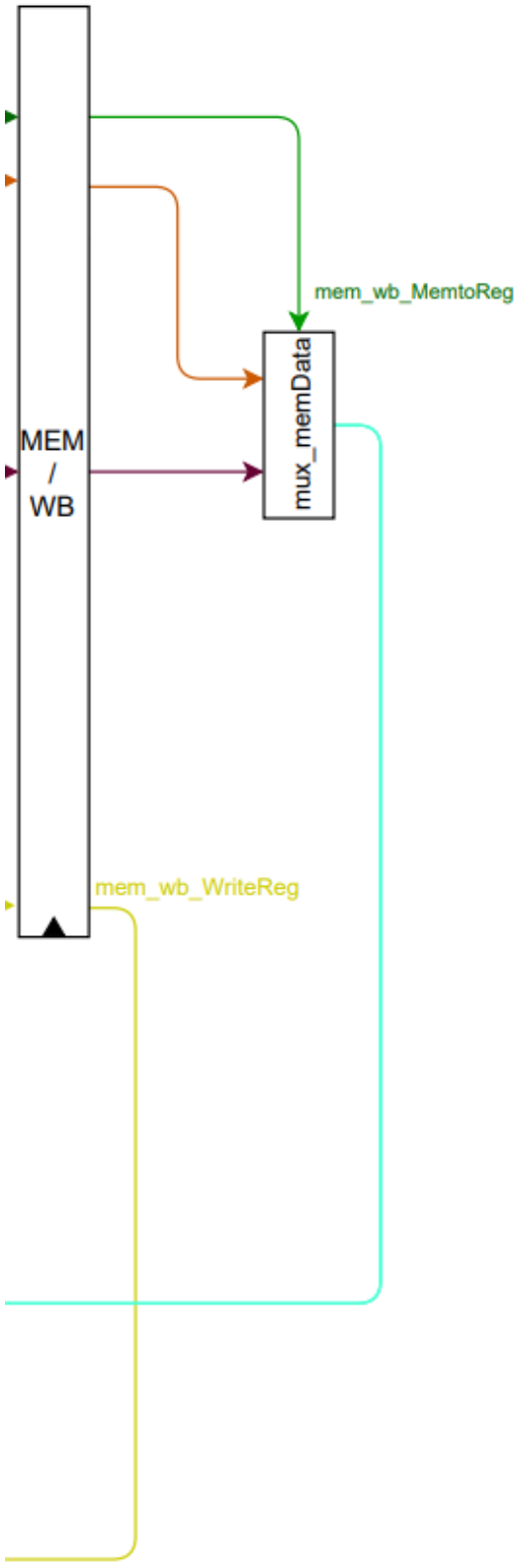
Het lezen uit het datamemory gebeurt altijd, de multiplexer MemData in de volgende stage zal bepalen of deze zal doorgekoppeld worden. Bij het scrijven op het memory of de stack wordt er gebruik gemaakt van 2D array registers in Verilog, voor de GPIO's worden er gewone registers gebruikt.

```
always @* begin
  Memory_select = 0; LEDR_select = 0; SW_select = 0; KEY_select = 0;
  HEX0_select = 0; HEX1_select = 0; HEX2_select = 0; HEX3_select = 0;
  HEX4_select = 0; HEX5_select = 0; HEXValue_select = 0;
  HEXMode_select = 0; Stack_select = 0;

  case(address[XLEN-1:12])
    {{(XLEN-32){1'b0}}, 20'h10000} : Memory_select = 1;
    {{(XLEN-32){1'b0}}, 20'h7ffff} : Stack_select = 1;
    {{(XLEN-32){1'b0}}, 20'h40000} : case (address[11:8])

                                0: LEDR_select = 1;
                                1: SW_select = 1;
                                2: KEY_select = 1;
                                3: case (address[7:0])

                                    8'h0:   HEX0_select = 1;
                                    8'h4:   HEX1_select = 1;
                                    8'h8:   HEX2_select = 1;
                                    8'hC:   HEX3_select = 1;
                                    8'h10:  HEX4_select = 1;
                                    8'h14:  HEX5_select = 1;
                                    8'h18:  HEXValue_select = 1;
                                    8'h1C:  HEXMode_select = 1;
                                    default: ;
                                  endcase
                                  default: ;
                              endcase
      default: ;
  endcase
end
```

Listing 3: Case in datamemory om het juiste deel van de memory te bepalen

# 7 WB stage

Write back (WB) is het vijde en laatste pipeline stage die aan bod komt. Hierin zal bepaald worden welke waarde er teruggekoppeld zal worden naar het register, zodat deze hierin geschreven kan worden. In figuur 11 is deze stage te zien.




Figuur 11: Schematise weergaven van de Write back (WB) stage

## 7.1 MEM/WB register

In dit register komen de waardes van de ALU en het datamemory binnen. Deze worden op hun beurt doorgekoppeld naar de multiplixer MemData. Ook zal het controle signaal voor de multiplexer en het writeback register doorgekoppeld worden.

## 7.2 Mux MemData

Via het selectie signaal van deze multiplexer `mem wb MemToReg` zal er bepaalt worden welke waarde er zal worden teruggekoppeld. Dit is oftewel de waarde die uit uit het memory gelezen word of de waarde die de ALU berekent heeft.

# 8 Programma Knight Rider

```
#-------------------------------------------------------------
# Jeroen en Jonas
#
# Knight Rider
#-------------------------------------------------------------

# GPIO Base Addresses
.equ GPIO_BASE_ADDRESS 0x40000000

# Offsets for GPIO peripherals
.equ LEDR      0x000
.equ SW        0x100
.equ KEY       0x200
.equ HEX0      0x300
.equ HEX1      0x304
.equ HEX2      0x308
.equ HEX3      0x30C
.equ HEX4      0x310
.equ HEX5      0x314
.equ HEX_VALUE 0x318
.equ HEX_MODE  0x31C

.text

    li t0, 0x40000000       # memory mapped peripherals base address
    li s1, 0x00A00000       # 10 485 760 (max counter)
    li s2, 0x00000001       # initialize rechter led
    li s3, 0x00000000       # boolean links of rechts (naar links = 0)
    li s4, 0x00000200       # Waarde meest linkse led
    li s5, 0x00000001       # Waarde meest rechtse led
    li s6, 0                # aantal keer knight rider (1 periode)
    sw s2, LEDR(t0)         # initialize leds
    sw s5, HEX_MODE(t0)     # initialize hexmode
    sw s6, HEX_VALUE(t0)    # initialize hex_value



    startcounter:
    li a0, 0                # counter op 0
    counter_loop:
    lw s7, SW(t0)           #lezen van de switch value

                            # for (i = 0; i < max counter; i++) {
    bge a0, s1, exit_loop1  # i < max counter
    add a0, a0, s7          # i = i + switch value
    j counter_loop
```

```
exit_loop1:
beq s3, zero, shiftleft    # kijk of je naar links of recht gaat
beq s2, s5, changeToLeft   # wanneer op meest linkse led => verander van
                           # richting
srli s2, s2, 1             # 1 led opschuiven naar rechts
sw s2, LEDR(t0)            # store in LEDR
j startcounter             # TERUG NAAR COUNTER

shiftleft:
beq s2, s4, changeToRight # wanneer op meest rechtse led => verander van
                          # richting
slli s2, s2, 1            # 1 led opschuiven
sw s2, LEDR(t0)           # store in LEDR
j startcounter            # TERUG NAAR COUNTER

changeToRight:
addi s3, s3, 1            # BOOLEAN OP 1
j startcounter            # TERUG NAAR COUNTER

changeToLeft:
addi s6, s6, 1            # PERIODE++
addi s3, s3, -1           # BOOLEAN OP 0
sw s6, HEX_VALUE(t0)      # WAARDE SCHRIJVEN NAAR HEX_VALUE
j startcounter            # TERUG NAAR COUNTER
```