Processor Design Using Verilog

Omkar Miniyar(201530217)

Ayush Malpani(201530241)

Mentor: Prof Lavanya Ramapantulu

Stage 1 : SIMPLE UNICORE 32 BIT PROCESSOR

INSTRUCTION FORMAT :

All instructions are fixed 32 bits in length and must be aligned on a four-byte boundary in memory. We try to keep the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding.In particular, the sign bit for all immediate instructions are always in bit 31 of the instruction to speed sign-extension circuitry.
Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective byte address is obtained by adding register rs1 to the sign-extended 12-bit offset. Loads copy a value from memory to register rd. Stores copy the value in register rs2 to memory.

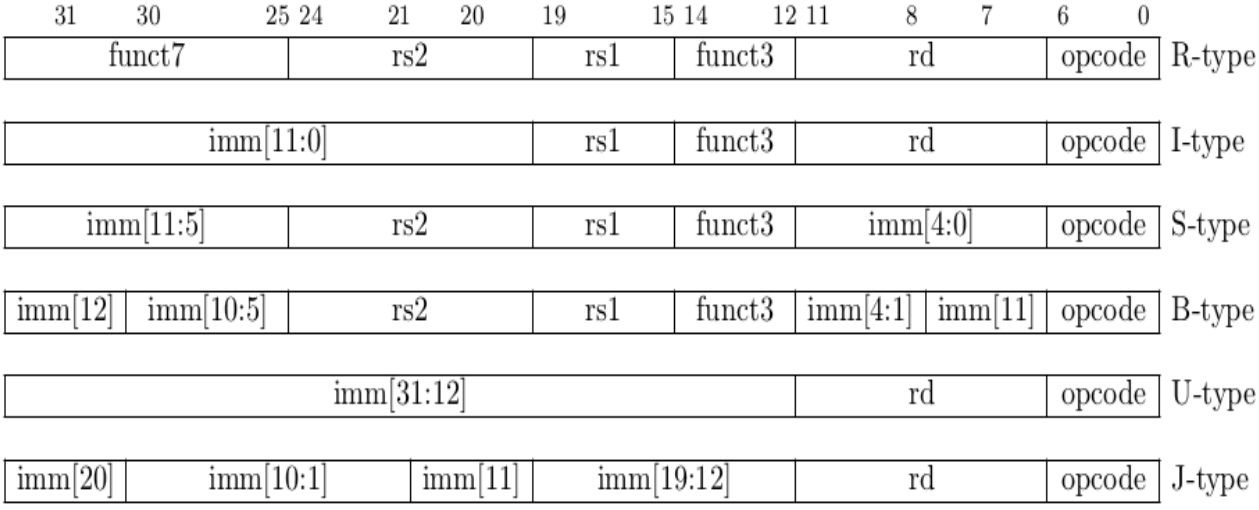


Fig 1 – Instruction formats showing immediate values

TYPES OF IMMEDIATE EXTENSIONS :

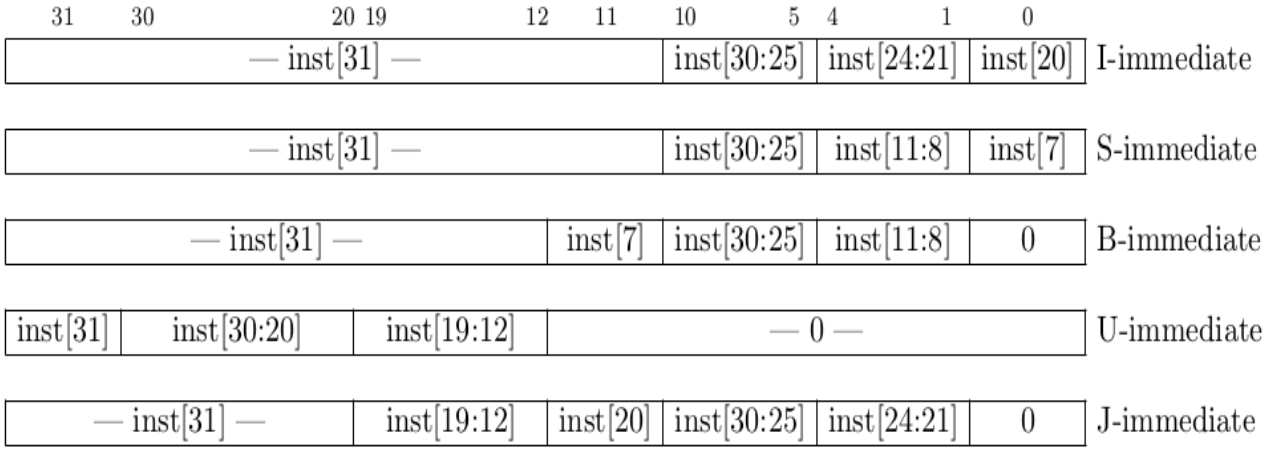


Fig2 – Types of Immediate produced. The fields are labeled with the instruction bits used to construct their value

Conventions used in table given below:

R[rx] : general-purpose register value for register specifier rx

sext : sign extend to 32 bits

M_4B[addr] : 4-byte memory value at address addr

PC : current program counter

<s : signed less-than comparison

>=s : signed greater than or equal to comparison

<u : unsigned less-than comparison

>=u : unsigned greater than or equal to comparison

imm : immediate according to the immediate type

DIFFERENT OPERATIONS:

Instruction	Assembly	Description	Instruction Format
REGISTER-REGISTER IMMEDIATE INSTRUCTIONS			
ADD	add rd, rs1, rs2	R[rd] = R[rs1] +R[rs2]	R
SUB	sub rd, rs1, rs2	R[rd] = R[rs1] - R[rs2]	R
AND	and rd, rs1, rs2	R[rd] = R[rs1] & R[rs2]	R
OR	or rd, rs1, rs2	R[rd] = R[rs1] | R[rs2]	R
XOR	xor rd, rs1, rs2	R[rd] = R[rs1] ^ R[rs2]	R
SLT	slt rd, rs1, rs2	R[rd] = (R[rs1]<sR[rs2])	R
SLTU	sltu rd, rs1, rs2	R[rd] = (R[rs1] <u R[rs2])	R
SRA	sra rd, rs1, rs2	R[rd] = R[rs1] >>> R[rs2][4:0]	R
SRL	srl rd, rs1, rs2	R[rd] = R[rs1] >> R[rs2][4:0]	R
SLL	sll rd, rs1, rs2	R[rd] = R[rs1] << R[rs2][4:0]	R
MUL	mul rd, rs1, rs2	R[rd] = R[rs1] * R[rs2]	R
REGISTER IMMEDIATE INSTRUCTIONS			
ADDI	addi rd, rs1, imm	R[rd] = R[rs1] + sext(imm)	I type, I immediate
SUBI	Subi rd,rs1,imm	R[rd] = R[rs1] - sext(imm)	I type, I immediate
ANDI	andi rd, rs1, imm	R[rd] = R[rs1] & sext(imm)	I type, I immediate
ORI	ori rd, rs1, imm	R[rd] = R[rs1] | sext(imm)	I type, I immediate
XORI	xori rd, rs1, imm	R[rd] = R[rs1] ^ sext(imm)	I type, I immediate
SLTI	slti rd, rs1, imm	R[rd] = (R[rs1] <s sext(imm))	I type, I immediate
SLTIU	sltiu rd, rs1, imm	R[rd] = (R[rs1] <u sext(imm))	I type, I immediate

SRAI	srai rd, rs1, imm	R[rd] = R[rs1] >>> imm	I type
SRLI	srli rd, rs1, imm	R[rd] = R[rs1] >> imm	I type
SLLI	slli rd, rs1, imm	R[rd] = R[rs1] << imm	R type
LUI	lui rd, imm	R[rd] = imm << 12	U type, U immediate
AUIPC	auipc rd, imm	R[rd] = PC + (imm << 12)	U type, U immediate
MEMORY ACCESS INSTRUCTIONS			
LW	lw rd, imm(rs1)	R[rd] = M_4B[R[rs1] +sext(imm)]	I type, I immediate
SW	sw rs2, imm(rs1)	M_4B[R[rs1] + sext(imm)] = R[rs2]	S type, S immediate
UNCONDITIONAL JUMP INSTRUCTIONS			
JAL	jal rd, imm	R[rd] = PC + 4; PC = PC + sext(imm)	U type, J immediate
JR	jr rs1	PC = R[rs1]	I type, I immediate
JALR	jalr rd, rs1, imm	R[rd] = PC + 4; PC = (R[rs1] + sext(imm))	I type, I immediate
BRANCH TYPE INSTRUCTIONS			
BEQ	beq rs1, rs2, imm	PC = (R[rs1] == R[rs2]) ? PC + sext(imm) : PC + 4	S type, B immediate
BNE	bne rs1, rs2, imm	PC = (R[rs1] != R[rs2]) ? PC + sext(imm) : PC + 4	S type, B immediate
BLT	blt rs1, rs2, imm	PC = (R[rs1] <s R[rs2]) ? PC + sext(imm) : PC + 4	S type, B immediate
BGE	bge rs1, rs2, imm	PC = (R[rs1] >=s R[rs2]) ? PC + sext(imm) : PC + 4	S type, B immediate
BLTU	bltu rs1, rs2, imm	PC = (R[rs1] <u R[rs2]) ? PC + sext(imm) : PC + 4	S type, B immediate

BGEU	bgeu rs1, rs2, imm	PC = (R[rs1] >=u R[rs2]) ? PC + sext(imm) : PC + 4	S type, B immediate

Table1 – different instructions present in processor, their assembly level code and instruction formats

INSTRUCTIONS	Summary
REGISTER – REGISTER ARITHMETIC INSTRUCTIONS	
ADD	Addition with 3 GPRs, no overlflow exception
SUB	Subtraction with 3 GPRs,no overflow exception
AND	Bitwise logical AND with 3 GPRs
OR	Bitwise logical OR with 3 GPRs
XOR	Bitwise logical XOR with 3 GPRs
SLT	Record signed less than comparison with 2 GPRs
SLTU	Record unsigned less than comparison with 2 GPRs
SRA	Shift right arithmetic by register value(sign extend)
SRL	Shift right logical by arithmetic value(append zeros)
SLL	Shift left logical by register value
MUL	Signed multiplication with 3 GPRs,no overflow exception
REGISTER – IMMEDIATE ARITHMETIC INSTRUCTIONS	
ADDI	Add constant, no overflow exception
SUBI	Subtract constant, no overflow exception
ANDI	Bitwise logical AND with constant
ORI	Bitwise logical OR, with constant
XORI	Bitwise logical XOR with constant

SLTI	Set GPR if GPR<constant, signed comparison
SLTIU	Set GPR if GPR<constant,unsigned comparison
SRAI	Shift right arithmetic by constant (sign-extend)
SRLI	Shift right logical by constant(append-zeros)
SLLI	Shift left logical constant(append zeros)
LUI	Load constant into upper bits of word
AUIPC	Load PC + constant into upper bits of word
Memory Instructions	
LW	Load word from memory
SW	Store word from memory
UNCONDITIONAL JUMP INSTRUCTIONS	
JAL	Jump to address and place return address in GPR
JR	Jump to address
JALR	Jump to address and place return address in GPR
CONDITIONAL BRANCH INSTRUCTIONS	
BEQ	Branch if 2 GPRs are equal
BNE	Branch if 2 GPRs are not equal
BLT	Branch based on signed comparison of two GPRs
BGE	Branch based on signed comparison of two GPRs
BLTU	Branch based on unsigned comparison of two GPRs
BGEU	Branch based on unsigned comparison of two GPRs

Table2 – Different instructions present and summary of task that needs to be done for each instructions

There will be total 37 instructions present in our processor. Out of which 2 (LW,SW) will be for memory access, 3 (JAL,JR,JALR) will be for unconditional branch instructions, 6(BEQ,BNE,BLT,BGE,BLTU,BGEU) will be for conditional branch instructions rest of the instructions rest will be logical and arithmetic instructions.

In our register file there will be total of 31 GPRs x1 – x31, which holds integer value. Register x0 is hardwired to the constant zero. Each register is 32 bits wide.

x0 : zero the constant value 0

x1 : ra return address (caller saved)

x2 : sp stack pointer (called saved)

x3 : gp global pointer

x4 : tp thread pointer

x5 : t0 temporary registers (caller saved)

x6 : t1 "

x7 : t2 "

x8 : s0/fp saved register or frame pointer (callee saved)

x9 : s1 saved register (callee saved)

x10 : a0 function arguments and/or return values (caller saved)

x11 : a1 "

x12 : a2 function arguments (caller saved)

x13 : a3 "

x14 : a4 "

x15 : a5 "

x16 : a6 "

x17 : a7 "

x18 : s2 saved registers (callee saved)

x19 : s3 "

x20 : s4 "

x21 : s5 "

x22 : s6 "

x23 : s7 "

x24 : s8 "

x25 : s9 "

x26 : s10 "

x27 : s11 "

x28 : t3 temporary registers (caller saved)

x29 : t4 "

x30 : t5 "

x31 : t6 "

Caller Saved Registers: Used to hold temporary quantities that need not be preserved across calls.

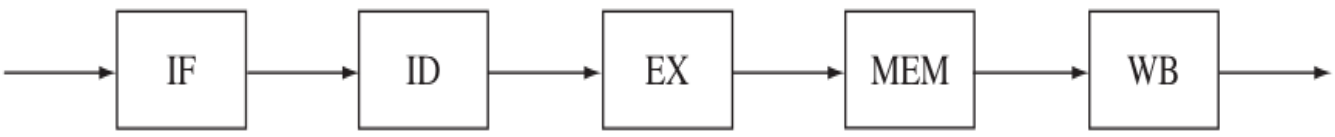
Callee Saved Register: Used to hold long lived values that should be preserved across calls.

Pipelining

Pipelining is a process of multitasking instructions in the same data path. An instruction can be loaded into the data path at every clock cycle, even though each instruction takes up to 5 cycles to complete. At every clock cycle, each instruction is stored into a different stage in the pipeline. Doing this does not affect any of the other instructions in the processor because they are in different stages. Each stage contains a different part of the data path and has a specific function. This allows the user of the processor to fully utilize all components of the data path simultaneously, causing an increase in the throughput of their program.

We will be implementing five stage pipelined processor -

- Instruction fetch (IF)
- Instruction decode / register fetch stage (ID)
- Execution stage (Ex)
- Memory access stage (MEM)
- Write back stage (WB)



In IF stage, reads an instruction from memory at the address pointed by PC. Concurrently in this stage PC is incremented to point to next instruction to be executed.

On next clock cycle, instruction enters ID stage, primary action is to decode opcode to determine what operation instruction will perform. The bits that specify the register operands are used to read operands from register. Branch target address is also computed in this stage. Any values that are read or computed unnecessarily are simply ignored and not used in subsequent pipeline stages.

After being decoded in ID stage, instruction advances to Ex stage. Opcode has been decoded by this stage, operand values have been read form registers. Information is passed from ID stage and used to set up the ALU to compute desired operand during this Ex stage if instruction is reg – reg ALU operation such as AND,ADD etc. If instruction is reg-immediate operation, sign extended ALU operation is used as one input to ALU with other input being one of the operand values read in decode stage.Now that desired operation has been computed in Ex stage, or effective memory address has been determined, these values are passed to MEM stage.

MEM stage is dedicated to accessing memory for load and store instructions. The effective address computed in Ex stage is used to read desired location from memory for load instructions. For store instructions operand value read in ID stage is written to memory at given address.

Final stage in pipeline, WB is where value computed Ex are finally written back to destination register. The address of this reg was previously determined in ID stage and has been passed through all of intermediate stages. The value that is written back is either value read from memory, if instruction is load or value computed in Ex stage. Store instruction do nothing in this stage since they have already completed their operation in MEM stage.

Dependencies and Hazards -

Dependencies between instructions that try to execute in the pipeline at the same time can lead to hazards in the pipeline.

There are two fundamental types of dependencies that arise in programs, control dependencies and data dependencies.

Control dependencies -

instruction x is preceding instruction y in the program

A control dependence exists from instruction x to y when x is a branch instruction whether or not y is executed depends on whether branch instruction x is taken or not.

Pipeline must be able to detect branch instructions and ensure instruction that follow branch are executed in pipeline only when branch direction is known.

Flow Dependencies – Exists from instruction x to instruction y when y needs value produced by x before y can begin executing.

Pipeline Hazards -

Structural hazard – Both instructions in the pipeline try to use the same resource. We will be designing such that this type of hazard do not occur in our processor. It is a common problem that occurs because an instruction must be fetched from memory on every clock cycle. Additionally there could be load or store instructions in MEM stage trying to read or write at the same time. Because of this, we will be dividing top level cache into two parts instruction cache and data cache. The instruction cache is accessed on every clock cycle to feed instructions into the pipeline. Any load or store then can access seperate data cache in the memory stage without interfering with instruction fetch stage.

Branch Hazard – A branch hazard occurs because in most pipeline implementations, outcome of branch is not known until several cycles after branch instructions has been fetched into pipeline. If branch ultimately turns out to be not taken, these instructions continue their execution through the entire pipeline as normal. If branch turns out to be taken, these instructions should not be executed since they are not taken path of the branch.

Unfortunately determining outcome of the branch instruction cannot be completed until the decode stage. As a result, once pipeline detects branch direction is taken, pipeline's branch logic must flush the incorrect instructions out of the pipeline and begin fetching the instructions located at branch target address. It must reset PC to begin fetching instructions from branch location.

We will be using NOP instruction to deal with branch hazard. The nop instruction that gets inserted introduces one cycle delay after every taken branch.

Data Hazard -

Read after Write (RAW) – Problem caused by RAW hazard is that result value produced by most instructions is not written to register until the write back stage, which is the last stage of the pipeline. However instructions read their operands from the register early in the pipeline in the ID stage. To deal with this type of hazard we will be using data forwading.

WAR (Write After READ) – This type of hazard will not be occuring in our processor design. This type of hazard occurs when instruction scheduling is done.

HAZARD DETECTION UNIT -

It is used to check for potential hazards, and to determine when pipeline is to be stalled, when bubble is to be introduced and when data must be forwaded in each stage of pipeline. Each stage of the pipeline consists of several MUXs and hazard detection unit generates control inputs for these MUXs.

Convention used below

Instruction 1 (instruction preciding instruction 2)

Instruction 2 (instruction following instruction 1)

1] arithmetic(R)-arithmetic(R) :

add rd,r1,r2 (Value of r1+r2 is stored in 5th stage in rd but we will have value of r1+r2 in temp register after 3rd stage)

Sub r3 r2 rd (value of rd is required in 3rd stage, we will directly get it by forwarding from temp. register)

2] arithmetic(I)-arithmetic(I) :

Addi rd rs1 imm1

Subi r2 rd imm2

(Same as above)

3] MEMORY - MEMORY :

LW rd imm(rs1) (Value of rd is obtained after 4th stage)

SW rd imm(rs2) (We need rd at the start of 4th stage so it can be done by forwarding from temp register)

LW rd imm(rs1) (Value of rd is obtained after 4th stage)

SW r2 imm(rd) (Stall + forwarding)

SW rd imm(rs1)

LW rd imm(rs2) (No problems in this case)

4] Memory- Branch :

Lw rd imm(rs1) (Value of rd is written back after 5th stage but we can forward its value after 4th stage from temp register)

Beq rs1 rd imm (Value of rd is required in 3rd stage so we need one stall here)

(So one stall and one forwarding is required in this case.)

Beq rs1 rs2 imm
Lw rd imm(rs1)
(No dependencies)

5] Branch - Branch :

Beq rs1 rs2 imm
Bge rs1 rs2 imm
(No relations between instructions here)

6] **R- Memory :**

Add rd rs1 rs2 (Value in rd is written after 5th stage but we can forward it using temp register after 3rd stage)
Sw rd imm(rs1) (We need rs1 at the start of 4th stage so no dependencies)

Sw rd imm(rs1)
Add rd rs1 rs2
No dependencies here also

Add rd rs1 rs2
Sw rs2 imm(rd) (We need rd at the start of 3rd cycle so we will use forwarding)

Lw rd imm(rs1) (We will get rd after 4th stage by using temp register and forwarding)
Add r3 rd rs2 (We need rd at the start of 3rd cycle so we need one stall)

Add rd rs1 rs2
Lw r3 imm(rd) (Only forwarding is sufficient)

7] R - I:
Add rd rs1 rs2
Addi rs2 rd imm
(Forwarding is required)

Addi rd rs1 imm
Add rs1 rd rs2 (Forwarding is required)

8] R - Unconditional :

Add rd rs1 rs2
JAL rd imm
(No relation)

Add rd rs1 rs2
JR rd (Forwarding is required)

Add rd rs1 rs2
JALR rs3 rd imm (Forwarding is required)

JALR rs3 rd imm
Add rd rs1 rs2
(No relation)

9] R- Branch :

Add rd rs1 rs2
Beq rd rs3 imm (Forwarding is required)

Beq rd rs3 imm
Add rd rs1 rs2
(No relation)

10] Unconditional- Memory :

Lw rd imm(rs1) (We will get rd after 5th stage but using temp register we can forward after end of 4th stage)
Jr rd (We need rd at the start of 3rd stage so we need one stall as well)

Sw rd imm(rs1)
Jr rd
(No relation)

Lw rd imm(rs1)
JAL rd imm
(No relation)

Lw rd imm(rs1)

JALR rs2 rd(imm) (One stall and one forwarding)

JALR

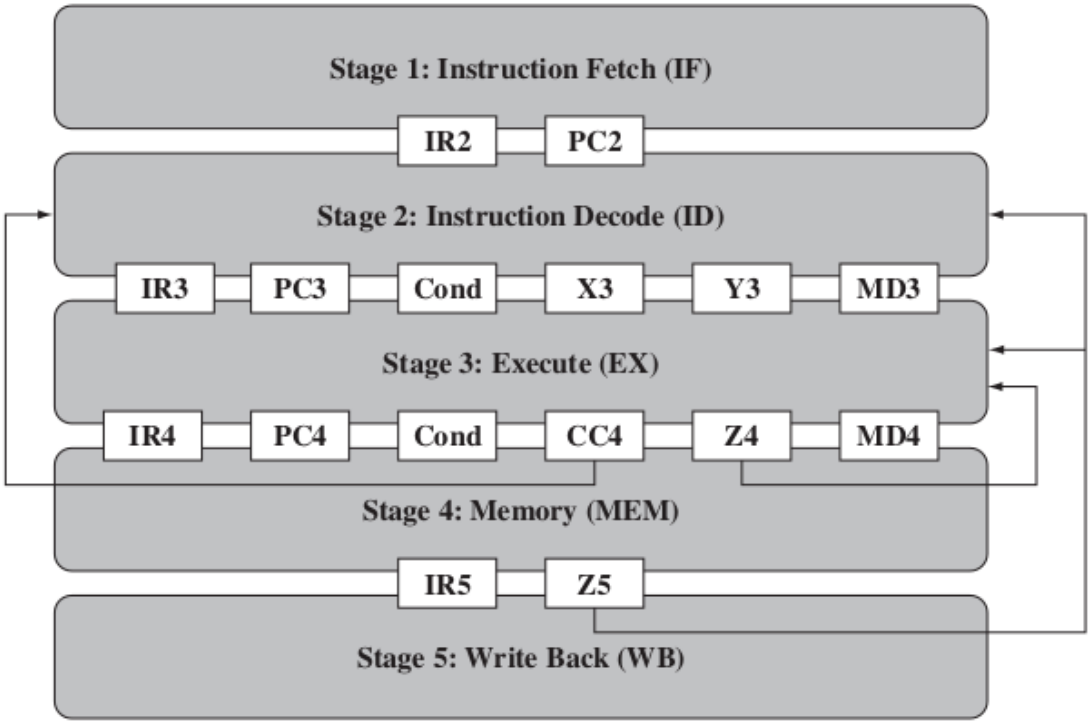
LW **(DOUBT)**

12] Unconditional- Branch : [Doubt]

12] Unconditional – Unconditional [Doubt] – When unconditional jump is executed should next two instructions ignored or not ?

Stall + forward	**Only forward**
Lw + JALR(mem+uncond-j)	Add + Sub (arith + arith)
Lw + JR(mem + uncond-j)	Add + Subi (arith + arithi)
Lw + Add(mem+arith)	Lw + Sw (mem + mem)
Lw + Beq(mem + branch)	Add + Sw (arith + mem)
Lw + Sw(mem + mem)	Add + Lw (arith + mem)
	Add + Addi (arith + arithi)
	Addi + Add (arithi + arith)
	Add + JR (arith + uncond-j)
	Add + JALR(arith+uncond-j)
	Add + Beq (arith + branch)

OVERVIEW OF PIPELINE STRUCTURE -



From the ID stage to the EX stage, a larger number of intermediate registers is introduced. As before, we propagate the instruction and program counter, this time through IR3 and PC3. As we will see later, this includes mechanisms to handle pipeline stalls and bubbles. Notably, any pipeline stalls are performed at this stage.

During memory write instructions, the ALU inputs may be used for address computation, and hence the data to be written is stored in the MD3 register. The EX stage propagates the condition bit from the ID stage, and also has an IR4 and PC4 for the instruction register and the program counter, respectively. As before, these may be configured for introducing pipeline bubbles, but configuration for pipeline stalls is not necessary since these are processed in the ID stage.

Beside these registers, the memory write data is stored in the MD4 register, the ALU results are stored in the Z4 register, and the condition codes in the CC4 register. Finally, the MEM stage only propagates the instruction register to IR5, and the register write data to the Z5 register. Data forwading is implemented by feeding back the Z4, Z5 and CC4 bits to various pipeline stages.