MIPS PIPELINED PROCESSOR

Abdelrahman Mohamed Abdelnabi
3466

Ahmed Lotfey Atia Siam
4129

Ahmed Mohamed Fayez
4130

November 30, 2019

Abstract

Pipelining is a powerful way to improve the throughput of a digital system. We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is almost five times faster. Hence, the latency of each instruction is ideally unchanged, but the throughput is ideally five times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Pipelining introduces some overhead, so the throughput will not be quite as high as we might ideally desire, but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined. In this Lab, we will design and implement a MIPS piplined processor using *System Verilog*.

1 Introduction

This is a 5 stage (Fetch, Decode, Execute, Memory, Write Back) MIPS pipelined processor.

To ease the modification process and allow readers to see how we went on to modify the source code to implement a new instruction, we use *git* as a *version control system*. After most of the instructions, there will be a command that you can run, after *cloning the project from git*, to see the changes that we made at each step.

You can clone the project using this command:
`git clone https://github.com/abdelrahmanabdelnabi/MIPS-Pipelined-Processor.git`

alternatively, if you don't have git, you can to this link to see the project through its stages (i.e *commits*) on *Github*:
`https://github.com/abdelrahmanabdelnabi/MIPS-Pipelined-Processor`

1.1 Supported Instructions

Opcodes	Example Assembly	Semantics
add	add $1, $2, $3	$1 = 2 + $3
sub	sub $1, $2, $3	$1 = 2 - $3
add immediate	addi 1, $2, 100	$1 = 2 + 100
multiply	mult 2, $3	hi, lo = $2 * $3
divide	div $2, $3	lo = 2/$3, hi = $2 mod $3
move from hi	mfhi $1	$1 = hi
move from low	mflo $1	$1 = lo
and	and $1, $2, $3	$1 = $2 & $3
or	or $1, $2, $3	$1 = $2 | $3
shift left logical	sll $1, $2, 10	$1 = $2 << 10
shift right logical	srl $1, $2, 10	$1 = $2 >> 10
load word	lw $1, $2(100)	$1 = ReadMem32($2 + 100)
store word	sw $1, $2(100)	WriteMem32($2 + 100, $1)
load byte	lb $1, $2(100)	$1 = SignExt(ReadMem8($2 + 100))
store byte	sb $1, $2(100)	WriteMem8($2 + 100, $1)
branch on equal	beq $1, $2, Label	if ($1 == $2) goto Label
branch on not equal	bne $1, $2, Label	if ($1 != $2) goto Label
set on less than	slt $1, $2, $3	if ($2 < $3) $1 = 1 else $1 = 0
set on less than immediate	slti $1, $2, 100	if ($2 < 100) $1 = 1 else $1 = 0
jump	j Label	goto Label
jump register	jr $31	goto $31
jump and link	jal Label	$31 = PC + 4; goto Label

1.2 Initial Version

We will use the MIPS pipelined version developed in our textbook as the initial version which we will build on it other components to support the remaining instructions.

This version already supports the instructions:add (`add`), sub (`sub`), add immediate (`addi`), and (`and`), or (`or`), load word (`lw`), sw (`sw`), branch on equal (`beq`), set on less than (`slt`), jump (`j`), with a hazard and control unit.

The initial version also uses early branch resolution to make a branch decision in the decode stage.

1.3 fixing a small error

The version in the textbook uses an equal comparator in the decode stage for early branch resolution. The datapath module instantiates an `eqcmp` object which is not defined in any file. we implemented the module ourselves. You can find the source code of the module in appendix A.12

The pipelined processor writes the register file in the first half of the write-back stage and reads it the second half of the decode stage. However, this was not the case in the text book code. The register file was written and read at the same time. To fix this, we added a delay of 5 time units after the write.

After this two fixes the test bench indicates a successful simulation.

2 Extending the Design

In this section we will modify and/or extend the design of the MIPS pipelined processor to support new instructions as indicated in section 1.1

2.1 Branch on Not Equal

supporting this instruction is relatively easy, as branch on equal is already supported. We just have to detect the not equal condition. To do this, we modify the control unit to detect the opcode of `bne` and generate the required control signals.

We add a new control signal, `bneD` which is 1 when the instruction in the decode stage is a `bne`. all the control signals for `beq` and `bne` are same except for the signals `branchD` and `bneD`, their values are reversed.

The logic for `pcsrcD` is now different. `prcsrcD` is now 1 when either of the conditions of `beq` and `bne` are satisfied. i.e `pcsrcD = (branchD & equalD) | (bneD & ~equalD)`

The logic for stalling will need to detect the `bne` instruction and stall as necessary exactly the same like for `beq`. Therefore the hazard unit takes a new input which the control signal `bneD`.

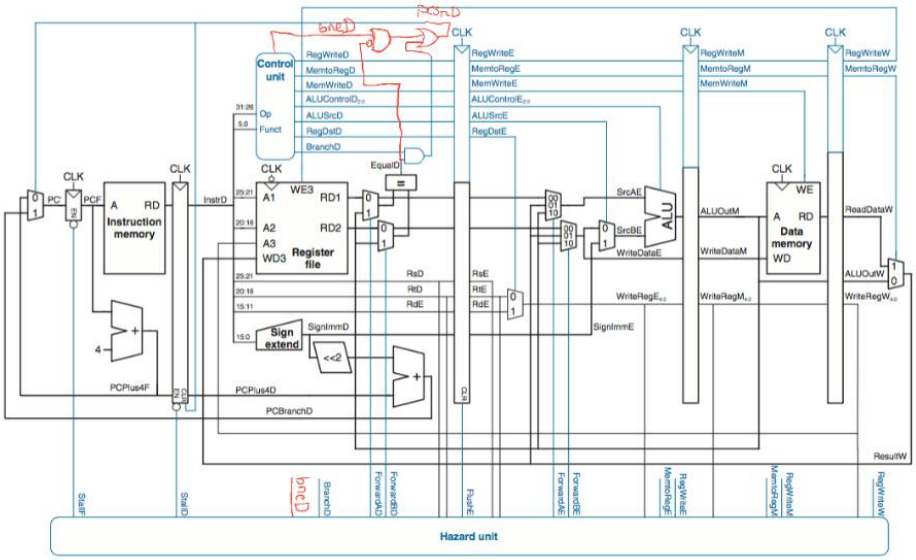

Figure 1: Pipelined processor with support for `bne`

This completes all the necessary changes required for the control unit and datapath. Figure 1 shows all the changes made to the data path.

We test the instruction by adding a `bne $s2, $2, 0x30` after the first instruction of our test program. Testing shows that the stall and branching works properly both when the branch is taken and not taken.

To see the changes to the source code to support this instruction run:
`git diff abdc c7c0`

2.2 load byte

An example of this instruction is `lb $s1, 40($s0)`, which loads the byte at the address of 40 + reg($s0), sign-extends it and puts it in register $s1.

To support this instruction we need to select a byte out of the 4 bytes read from memory each cycle according to $memaddr_{[1:0]}$, and then sign extend it to 32 bits. We will need a control signal that tells us that this is a `lb` instruction, then at the memory stage, we will select either the 32 bits memory output or the sign extended byte according to this

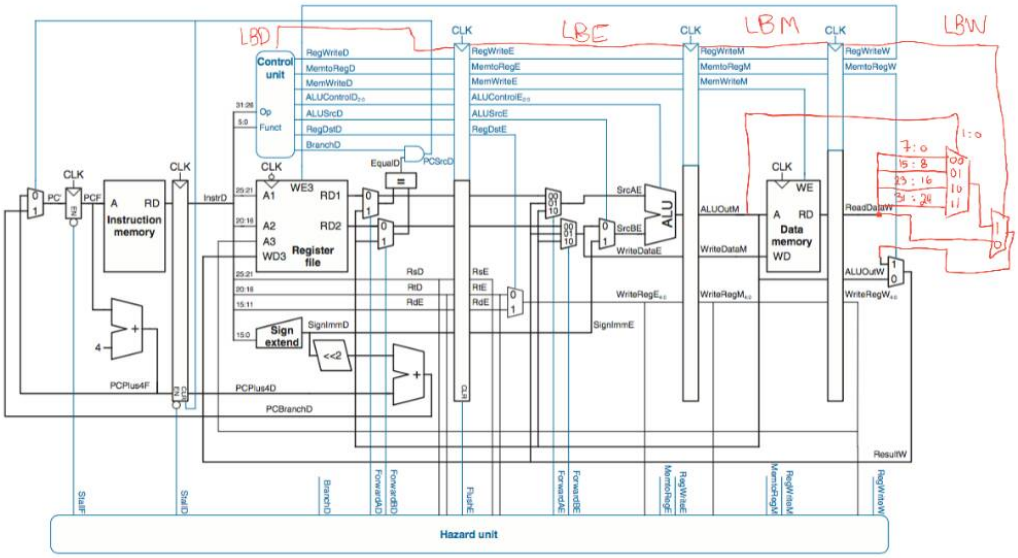

Figure 2: Pipelined processor with support for `lb`

signal. This implies using a 2-1 multiplexer that has this control signal as its selector. We will need another multiplexer to select the proper byte from the memory word read. Figure 2 shows the modifications made to the data path.

We test the instruction by modifying our test program to use `lb` instead of `lw` (temporarily). Since this should still the load 7 into register $s2, it won't modify the final result. We run the simulation and it indicates successful completion. we also see that the `lbW` signal takes the value 1 only at the write-back stage of the load byte instruction.

To see the changes made to the code to support this instruction run:
`git diff 18dc 5547`

2.3 Logical Shift

logical shift left, `sll` and right `srl` are R-Type instructions. The infrastructure for R-Type instructions already exists and therefore no significant changes to the datapath are required. Most of the change occurs because of the *shamt* field needs to be passed to the execute stage and the ALU. We add the shamt to the execute stage pipeline register, and modify the ALU to accept it as an input. The ALU decoder will need to detect the

$ALUControl_{3:0}$	Function
0000	A AND B
0001	A OR B
0010	A + B
0011	not used
1000	A AND $\overline{B}$
1001	A OR $\overline{B}$
1010	A - B
1011	SLT
0100	B $<<$ shamt
0101	B $>>$ shamt

Table 1: ALU control truth table

shift instructions and generate the proper ALUcontrol signals, so we add 2 rows to the decoder's truth table, one for each instruction. Moreover we extend the ALUControl signal to 4 bits to be able to encode the two new instructions. Table 1 shows new the ALU control signals truth table.

To test the two new instructions, we add `sll $s2, $s2, 4` and `srl $s2, $s2, 4` just before the last instruction in out test program. This won't modify the final result since $s2 contains 7 before running the shift instructions and we shift left then right by the same amount. Running the simulation, the correct result is written and completes successfully.

To see the changes made to the code to support these instructions run:
`git diff 2656 5ab4`

2.4 Jump and Link

The semantics of `jal label` is `$ra = PC + 4, goto label`. `jal` does exactly the same as jump but writes the value of PC onto register 31, the return address register.

Implementing this instruction won't affect the hardware as much because `jal` is mixture of jump and R-Type, both of which are already supported. We will add to the datapath a new mux that selects which data to write to the register file. The new mux will choose between *ResultW* and *PCPlus4W*, according to the selector *jal*, which is a new control signal that is true only if the instruction is `jal`. The *PCPlus4* signal must

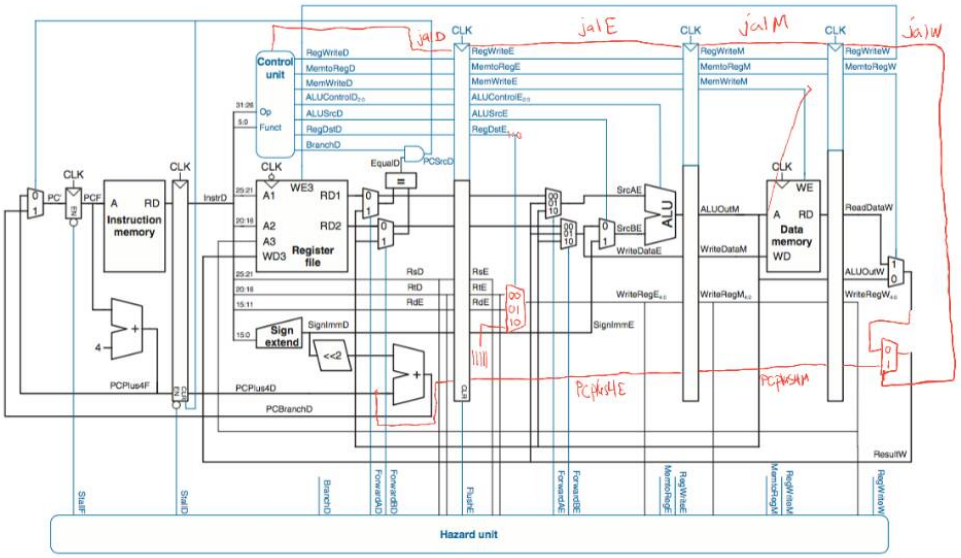

Figure 3: Pipelined processor with support for `jal`

be passed from the decode to the write back stage so that we can write it to the register file. We will also extend the *RegDst* mux, which selects which register to write on, to accept the number 31 as a new input. This means that we need to extend the mux selector control signal, *RegDst*, to 2 bits.

The main decoder sets the *Jump* signal to 1 so that the hazard unit handles flushes appropriately. Also no modifications to the hazard unit are required since the writing to register file part of the instruction (*WriteRegE* and *RegWriteE* signals) is the same as for R-Type instructions. Figure 3 shows the modifications made to the data path.

To test the instruction, we modify the test program to use `jal` instead of `j` on the instruction at address 3C. This won't change the results. It will only write the value of PC + 4, which is 40 at the time of executing the instruction, to the return address register, $ra.

Figure 4 shows the simulation of the test program with `jal` being used. Notice the simulation ends successfully the *jalD* and *jalW* signals are 1 when the instruction is decoded and when the result is written to the register file. Also notice that the return address register has the correct value (0x40) written to it, at the second half of the write back stage of the `jal` instruction.

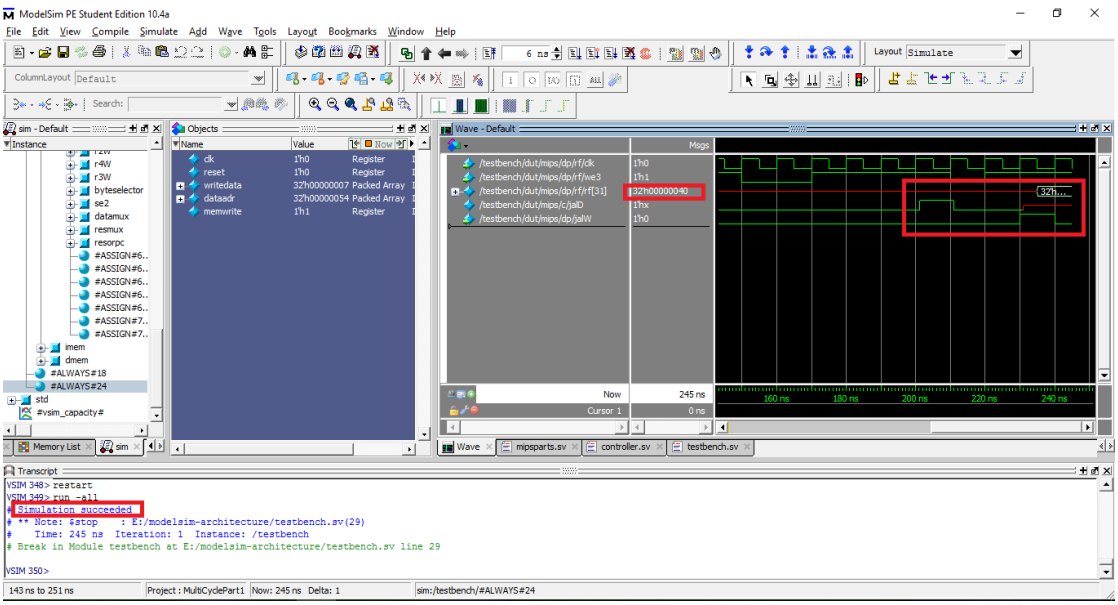

Figure 4: simulation and testing of `jal`

To see the changes made to the code to support this instruction run:
`git diff 8f6f e0ea -- mips.sv controller.sv datapath.sv maindecoder.sv`

2.5 Multiplication and Division

As the textbook states, Multiplication and division are somewhat different from other arithmetic operations. Multiplying two 32-bit numbers produces a 64-bit product. Dividing two 32-bit numbers produces a 32-bit quotient and a 32-bit remainder.

The MIPS architecture has two special-purpose registers, hi and lo, which are used to hold the results of multiplication and division. `mult $s0, $s1` multiplies the values in $s0 and $s1. The 32 most significant bits of the product are placed in hi and the 32 least significant bits are placed in lo. Similarly, `div $s0, $s1` computes $s0/$s1. The quotient is placed in lo and the remainder is placed in hi.

2.5.1 The Multiplication and Divison unit

The normal ALU doesn't support multiplication and division operations. In reality, multiplication and division are executed using dedicated expensive circuits and usually take several clock cycles to finish. Our Design will be simple however. We will add a new unit that operates in the execute stage alongside the ALU to carry out multiplication and division.

The unit's inputs are the two numbers, which are exactly the same as those provided to the ALU, and a flag that tells the unit to either multiply or divide. The outputs are two 32-bit numbers, the hi and lo values. Appendix A.5 shows the unit's code.

2.5.2 Hi and Lo Registers

The hi and lo registers are used to support 64-bit multiplication and divsion results. They exist outside the register file as separate registers. However, we write them in the first half of the write back cycle and read them in the second half of the decode cycle. The registers are encapsulated in module with a write enable signal. It has two input ports to input the data to be written, and two output ports that has the data read.

2.5.3 Supporting `Mult` and `div` Instructions

Once we add the multiplication and division unit and the hi and lo registers to the datapath, it's only the addition or few control signals that's remaining. We add two new control signals, *multordiv* and *hlwrite*, the former tells the unit to multiply or divide, and the latter tells the hi and lo module to write the data at its input port to the registers. The control signals and the result of multiplication unit will be passed through the pipeline registers to the write back stage as usual (except for *multordiv* which is needed only in the execute stage).

Two more rows are added to the main decoder to detect `mult` and `div` and generate the needed control signals.

2.5.4 Supporting `mfhi` and `mflo`

`mfhi` and `mflo` turn out to be simple to implement as they operate in the write back stage only. The data needed, i.e the hi or lo values, are only needed in the write back

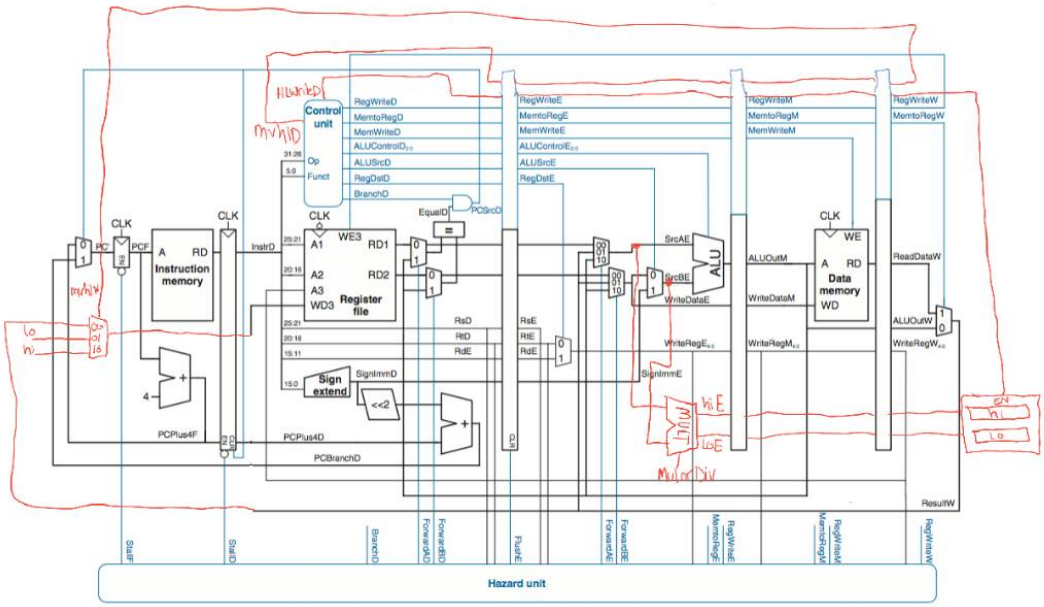

Figure 5: Pipelined processor with support for `mult`, `div`, `mvhi`, and `mflo`

stage, which eliminates the need for RAW hazard handling as no forwarding or stalling is needed.

To implement this instruction we need to select which value to write to the register file. We add a new mux that has the *ResultW*, hi, and lo as its inputs. The inputs are selected according to the instruction in execution, whether it's a `mfhi`, `mflo`, or any other instruction. Therefore we will need a 2 bit control signal, *mvhl*, for the multiplexer selectors.

Figure 5 shows all the modifications made to the data path to support `mult`, `div`, `mvhi`, and `mflo`.

To see the changes made to the source code to support these instructions, run:[1]
`git diff 2dce 306f`

[1] we forgot to add the *hiandlo* module to git at this commit, however you can still view the module starting from commit number `4503`

$ALUOp_{1:0}$	Operation
00	Add
01	Sub
10	Look at *funct*
11	SLT

Table 2: ALU operation Table

2.5.5 Hazard Handling

For `mult` and `div`, the operands for the instructions are properly already handled by the hazard unit since the forwarded operand is the one that the multiplication unit uses. For `mfhi` and `mflo`, the operands, i.e the hi and lo registers, are read at the memory cycle which guarantees that the proper value is read, as indicated in section 2.5.4.

2.6 Set If Less than Immediate

The ALU already supports `slt`, we just need to pass the immediate field as the second operand to the ALU. The datapath already has a path for passing the sign-extended immediate to the ALU, by setting *ALUSrc* to 1. We assign the *ALUOp* 11 to subtract B from A and set the result to 1 if A $<$ B and 0 otherwise. Table 2 shows the new ALUOp table.

2.7 Jump Register

`jr` jumps to the address stored in *reg(rs)*, instead of the JTA calculated from the immediate field of a jump instruction. Unlike the normal jump, `jr` is an R-Type instruction.

To Implement `jr`, we will extend the mux that selects which data gets written on the PC register. The extended mux will choose between PC' (the result of the previous mux), jump target address, and SrcAD (value of *reg(rs)*). We added a new control signal, *jr* that is true only if the instruction is `jr`. Now concerning the Selector of the extended mux , we will use the jump control signal as the least significant bit of the selector and the new *jr* control signal as the most significant bit. And of course the proper value of the selectors are generated by the main decoder. Figure 6 shows the modifications to the data path to implement this instruction.

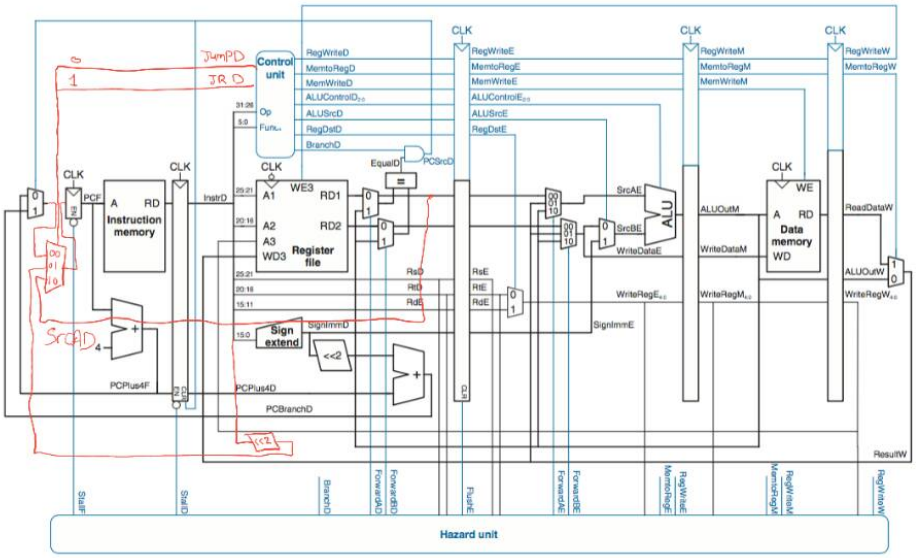

Figure 6: Pipelined processor with support for `jr`

2.8 Store Byte

According to the textbook and the final project description, store byte stores the least significant bits of a register in a byte in memory without affecting the remaining bytes in the same word in memory. This requires modifying the memory to able to write only one byte.

We add a pin to the memory that specifies if the memory should store only one byte of the data at its input port. The pin is connected to *sb*, a new control signal. Appendix A.8 has the modified source code for the memory module. Other than that, no more modifications are required. No changes are made to the data path.

To see the source code modifications made to support this instruction run:
`git diff b26a ea0a`

3 Final Main Decoder Table

Inst	Opcode	RegWrite	RegDst	ALUSrc	Branch	Bne	MemWrite	MemtoReg	Jump	Jal	Jr	Lb	Sb	MultorDiv	hlwrite	mvhl	ALUOp
R-Type	000000	1	00	0	0	0	0	0	0	0	0	0	0	0	0	00	00
JR	000000	0	00	0	0	0	0	0	0	0	1	0	0	0	0	00	00
MULT	000000	0	00	0	0	0	0	0	0	0	0	0	0	1	1	00	00
DIV	000000	0	00	0	0	0	0	0	0	0	0	0	0	0	1	00	00
MFLO	000000	1	01	0	0	0	0	0	0	0	0	0	0	0	0	01	00
MFHI	000000	1	01	0	0	0	0	0	0	0	0	0	0	0	0	10	00
LW	100011	1	00	1	0	0	0	1	0	0	0	0	0	0	0	00	00
SW	101011	0	00	1	0	0	1	0	0	0	0	0	0	0	0	00	00
BEQ	000100	0	00	0	1	0	0	0	0	0	0	0	0	0	0	00	01
BNE	000101	0	00	0	0	1	0	0	0	0	0	0	0	0	0	00	01
ADDI	001000	1	00	1	0	0	0	0	0	0	0	0	0	0	0	00	00
J	000010	0	00	0	0	0	0	0	1	0	0	0	0	0	0	00	00
LB	100000	1	00	1	0	0	0	1	0	0	0	1	0	0	0	00	00
SB	101000	0	00	1	0	0	1	0	0	0	0	0	1	0	0	00	00
JAL	000011	1	10	0	0	0	0	0	1	1	0	0	0	0	0	00	00
SLTI	001010	1	00	1	0	0	0	0	0	0	0	0	0	0	0	00	11

4 Simulation and Testing

To ensure that our modifications have been completed correctly and work together as intended, we test the completed pipelined processor on 3 programs, listed in appendix B.

Because there are too many signals to show and the program takes from 250 to 950 ns to execute, it would be cumbersome to include the simulation screenshots here as they it would take a lot of screenshots to show the simulation of one program. Therefore, its better if you could download the source files attached with this report, compile and run them and view the results on you screen. All the 3 programs have been tested and all show correct results.

A Source Code

A.1 datapath.sv

```
module datapath(input logic clk, reset,
input logic memtoregE, memtoregM, memtoregW,
input logic pcsrcD, branchD, bneD,
input logic alusrcE,
input logic [1:0] regdstE,
input logic regwriteE, regwriteM, regwriteW,
input logic jumpD, jalD, jalW, jrD, lbW,
input logic multordivE, hlwriteE, hlwriteM, hlwriteW,
input logic [1:0] mfhlW,
input logic [3:0] alucontrolE,
output logic equalD,
output logic [31:0] pcF,
input logic [31:0] instrF,
output logic [31:0] aluoutM, writedataM,
input logic [31:0] readdataM,
output logic [5:0] opD, functD,
output logic flushE);

logic forwardaD, forwardbD;
logic [1:0] forwardaE, forwardbE;
logic stallF;
logic [4:0] rsD, rtD, rdD, rsE, rtE, rdE, shamtD, shamtE
    ;
logic [4:0] writeregE, writeregM, writeregW;
logic flushD;
logic [31:0] pcnextFD, pcnextbrFD, pcplus4F, pcbranchD;
logic [31:0] signimmD, signimmE, signimmshD;
logic [31:0] srcaD, srca2D, srcaE, srca2E;
logic [31:0] srcbD, srcb2D, srcbE, srcb2E, srcb3E;
logic [31:0] pcplus4D, pcplus4E, pcplus4M, pcplus4W,
    instrD;
logic [31:0] aluoutE, aluoutW;
logic [31:0] readdataW, resultW;
logic zeroE;
logic [7:0] selectedbyteW;
logic [31:0] selectedbyteextW, selectedreaddataW;
logic [31:0] aluordata;
logic [31:0] hiE, loE, hiM, loM, hiW, loW;
logic [31:0] hireg, loreg, rfinput;

// hazard detection
hazard h(rsD, rtD, rsE, rtE, writeregE, writeregM,
writeregW,regwriteE, regwriteM, regwriteW,
memtoregE, memtoregM, branchD, bneD,
forwardaD, forwardbD, forwardaE,
forwardbE, stallF, stallD, flushE
);

// next PC logic (operates in fetch and decode)
mux2 #(32) pcbrmux(pcplus4F, pcbranchD, pcsrcD,
    pcnextbrFD);
mux3 #(32) pcmux(pcnextbrFD,{pcplus4D[31:28], instrD
    [25:0], 2'b00}, srcaD, { jrD,jumpD}, pcnextFD);

// register file (operates in decode and writeback)
regfile rf(clk, regwriteW, rsD, rtD, writeregW,
rfinput, srcaD, srcbD);

// lo and hi registers (operates ini decode and
    writeback)
hiandlo regs(clk, hlwriteW, hiW, loW, hireg, loreg);

// Fetch stage logic
flopenr #(32) pcreg(clk, reset, ~stallF, pcnextFD, pcF);
adder pcadd1(pcF, 32'b100, pcplus4F);

// Decode stage
flopenr #(32) r1D(clk, reset, ~stallD, pcplus4F,
    pcplus4D);
flopenrc #(32) r2D(clk, reset, ~stallD, flushD, instrF,
    instrD);
signext se(instrD[15:0], signimmD);
sl2 immsh(signimmD, signimmshD);
adder pcadd2(pcplus4D, signimmshD, pcbranchD);
mux2 #(32) forwardadmux(srcaD, aluoutM, forwardaD,
    srca2D);
mux2 #(32) forwardbdmux(srcbD, aluoutM, forwardbD,
    srcb2D);
eqcmp comp(srca2D, srcb2D, equalD);
assign opD = instrD[31:26];
assign functD = instrD[5:0];
assign rsD = instrD[25:21];
assign rtD = instrD[20:16];
assign rdD = instrD[15:11];
assign shamtD = instrD[10:6];
assign flushD = (pcsrcD & ~stallD) | jumpD | jrD;

// Execute stage
floprc #(32) r1E(clk, reset, flushE, srcaD, srcaE);
floprc #(32) r2E(clk, reset, flushE, srcbD, srcbE);
floprc #(32) r3E(clk, reset, flushE, signimmD, signimmE)
    ;
floprc #(32) r7E(clk, reset, flushE, pcplus4D, pcplus4E)
    ;
floprc #(5) r4E(clk, reset, flushE, rsD, rsE);
floprc #(5) r5E(clk, reset, flushE, rtD, rtE);
floprc #(5) r6E(clk, reset, flushE, rdD, rdE);
floprc #(5) shamtreg(clk, reset, flushE, shamtD, shamtE)
    ;
mux3 #(32) forwardaemux(srcaE, resultW, aluoutM,
    forwardaE, srca2E);
mux3 #(32) forwardbemux(srcbE, resultW, aluoutM,
    forwardbE, srcb2E);
mux2 #(32) srcbmux(srcb2E, signimmE, alusrcE, srcb3E);
alu alu(srca2E, srcb3E, alucontrolE, shamtE, aluoutE,
    zeroE);
multdivunit multiplier(srca2E, srcb2E, multordivE, hiE,
    loE);
mux3 #(5) wrmux(rtE, rdE, 5'b11111, regdstE, writeregE);

// Memory stage
flopr #(32) r1M(clk, reset, srcb2E, writedataM);
flopr #(32) r2M(clk, reset, aluoutE, aluoutM);
flopr #(32) r4M(clk, reset, pcplus4E, pcplus4M);
flopr #(32) r5M(clk, reset, hiE, hiM);
flopr #(32) r6M(clk, reset, loE, loM);
flopr #(5) r3M(clk, reset, writeregE, writeregM);

// Writeback stage
flopr #(32) r1W(clk, reset, aluoutM, aluoutW);
flopr #(32) r2W(clk, reset, readdataM, readdataW);
flopr #(32) r4W(clk, reset, pcplus4M, pcplus4W);
flopr #(32) r5W(clk, reset, hiM, hiW);
flopr #(32) r6W(clk, reset, loM, loW);
flopr #(5) r3W(clk, reset, writeregM, writeregW);

// load byte logic
mux4 #(8) byteselector(readdataW[7:0], readdataW[15:8],
    readdataW[23:16],readdataW[31:24],
aluoutW[1:0], selectedbyteW);
signext8 se2(selectedbyteW, selectedbyteextW);
mux2 #(32) datamux(readdataW, selectedbyteextW, lbW,
    selectedreaddataW);

mux2 #(32) resmux(aluoutW, selectedreaddataW, memtoregW,
     aluordata);
mux2 #(32) resorpc(aluordata, pcplus4W, jalW, resultW);
mux3 #(32) resulthilo(resultW, loreg, hireg, mfhlW,
    rfinput);
endmodule
```

A.2 maindecoder.sv

```
module maindec(input logic [5:0] op,
        output logic memtoreg, memwrite,
        output logic branch, bne, alusrc,
        output logic [1:0] regdst,
        output logic regwrite,
        output logic jump, jal, jr, lb, sb,
        output logic multordiv, hlwrite,
        output logic [1:0] mvhl,
        output logic [1:0] aluop,
        input logic [5:0] funct);
```

```
logic [18:0] controls;

assign {regwrite, regdst, alusrc, branch, bne,
        memwrite, memtoreg, jump, jal, jr, lb, sb,
        multordiv, hlwrite, mvhl, aluop} = controls;

always_comb
case(op)
        //6'b000000: controls <= 18'
             b1010_0000_0000_0000_10; //Rtype
        6'b000000:
        case(funct)
                6'b001000: controls <= 19'
                     b0_0000_0000_0100_0000_00; //JR
                6'b011000: controls <= 19'
                     b0_0000_0000_0000_1100_00; //MULT
                6'b011010: controls <= 19'
                     b0_0000_0000_0000_0100_00; //DIV
                6'b010010: controls <= 19'
                     b1_0100_0000_0000_0001_00; //MFLO
                6'b010000: controls <= 19'
                     b1_0100_0000_0000_0010_00; //MFHI

                default: controls <= 19'
                     b1_0100_0000_0000_0000_10; //Rtype
        endcase
        6'b100011: controls <= 19'
             b1_0010_0010_0000_0000_00; //LW
        6'b101011: controls <= 19'
             b0_0010_0100_0000_0000_00; //SW
        6'b000100: controls <= 19'
             b0_0001_0000_0000_0000_01; //BEQ
        6'b000101: controls <= 19'
             b0_0000_1000_0000_0000_01; //BNE
        6'b001000: controls <= 19'
             b1_0010_0000_0000_0000_00; //ADDI
        6'b000010: controls <= 19'
             b0_0000_0001_0000_0000_00; //J
        6'b100000: controls <= 19'
             b1_0010_0010_0010_0000_00; //LB
        6'b000011: controls <= 19'
             b1_1000_0001_1000_0000_00; //JAL
        6'b001010: controls <= 19'
             b1_0010_0000_0000_0000_11; //SLTI
        6'b101000: controls <= 19'
             b0_0010_0100_0001_0000_00; //SB
        default: controls <= 19'bxxxxxxxxxxxxxxxxxxx; //
             ???
endcase
endmodule
```

A.3 aludecoder.sv

```
module aludec(input logic [5:0] funct,
input logic [1:0] aluop,
output logic [3:0] alucontrol);
always_comb
case(aluop)
        2'b00: alucontrol <= 4'b0010; // add
        2'b01: alucontrol <= 4'b1010; // sub
        2'b11: alucontrol <= 4'b1011; // slti
        default: case(funct) // RTYPE
                6'b100000: alucontrol <= 4'b0010; // ADD
                6'b100010: alucontrol <= 4'b1010; // SUB
                6'b100100: alucontrol <= 4'b0000; // AND
                6'b100101: alucontrol <= 4'b0001; // OR
                6'b101010: alucontrol <= 4'b1011; // SLT
                6'b000000: alucontrol <= 4'b0100; // SLL
                6'b000010: alucontrol <= 4'b0101; // SRL
                default: alucontrol <= 4'bxxxx; // ???
        endcase
endcase
endmodule
```

A.4 controller.sv

```
module controller(input logic clk, reset,
        input logic [5:0] opD, functD,
        input logic flushE, equalD,

        output logic memtoregE, memtoregM,
        output logic memtoregW, memwriteM,
        output logic pcsrcD, branchD, bneD, alusrcE,
        output logic [1:0] regdstE,
        output logic regwriteE, regwriteM, regwriteW,
             jumpD,
        output logic jalD, jalW, jrD, lbW, sbM,
        output logic multordivE, hlwriteE, hlwriteM,
             hlwriteW,
        output logic [1:0] mfhlW,
        output logic [3:0] alucontrolE);

logic [1:0] aluopD;
logic memtoregD, memwriteD, alusrcD;
logic [1:0] regdstD;
logic regwriteD;
logic [3:0] alucontrolD;
logic memwriteE;
logic lbD, lbE, lbM, sbD, sbE;
logic jalE, jalM;
logic multordivD, hlwriteD;
logic [1:0] mfhlD, mfhlE, mfhlM;

maindec md(opD, memtoregD, memwriteD, branchD, bneD,
        alusrcD, regdstD, regwriteD, jumpD, jalD, jrD,
             lbD, sbD,
        multordivD, hlwriteD, mfhlD, aluopD, functD);

aludec ad(functD, aluopD, alucontrolD);

assign pcsrcD = (branchD & equalD) | (bneD & ~equalD);

// pipeline registers
floprc #(17) regE(clk, reset, flushE,
{memtoregD, memwriteD, alusrcD, regdstD, regwriteD,
      alucontrolD, jalD, lbD, multordivD, hlwriteD, mfhlD
      , sbD},
{memtoregE, memwriteE, alusrcE, regdstE, regwriteE,
      alucontrolE, jalE, lbE, multordivE, hlwriteE, mfhlE
      , sbE}
);

flopr #(9) regM(clk, reset,
{memtoregE, memwriteE, regwriteE, jalE, lbE, hlwriteE,
      mfhlE, sbE},
{memtoregM, memwriteM, regwriteM, jalM, lbM, hlwriteM,
      mfhlM, sbM}
);

flopr #(7) regW(clk, reset,
{memtoregM, regwriteM, jalM, lbM, hlwriteM, mfhlM},
{memtoregW, regwriteW, jalW, lbW, hlwriteW, mfhlW}
);

endmodule
```

A.5 multdivunit.sv

```
module multdivunit(input logic [31:0] a, b,
                input logic multordiv,
                output logic [31:0] hi, lo);

always_comb
if (multordiv)
        assign {hi, lo} = a * b;
else
        begin
        assign lo = a / b;
        assign hi = a % b;
        end

endmodule
```

A.6 hiandlo.sv

```
module hiandlo(input logic clk, we,
               input logic [31:0] wdhi, wdlo,
               output logic [31:0] rdhi, rdlo);

logic [31:0] hi, lo;
always
        begin
        if(clk)
            if (we)
                begin
                hi <= wdhi;
                lo <= wdlo;
                end
        #5;
        assign rdhi = hi;
        assign rdlo = lo;
        end
endmodule
```

A.7 alu.sv

```
module alu(input logic [31:0] A, B,
        input logic [3:0] F, // SRLV, XORI
        input logic [4:0] shamt, // SLL, SRL
        output logic [31:0] Y, output Zero);

logic [31:0] S, Bout;
assign Bout = F[3] ? ~B : B;
assign S = A + Bout + F[3];
always_comb
case (F[2:0])
        3'b000: Y <= A & Bout;
        3'b001: Y <= A | Bout;
        3'b010: Y <= S;
        3'b011: Y <= S[31];
        3'b100: Y <= S << shamt;
        3'b101: Y <= S >> shamt;
endcase
assign Zero = (Y == 32'b0);
endmodule
```

A.8 mipsmem.sv

```
module imem(input logic [5:0] a,
        output logic [31:0] rd);
        logic [31:0] RAM[4095:0];
initial
begin
        $readmemh("memfile.dat",RAM);
end

assign rd = RAM[a]; // word aligned
endmodule

module dmem(input logic clk, we, sbyte,
        input logic [31:0] a, wd,
        output logic [31:0] rd);

reg [31:0] RAM[4095:0];
initial
begin
        $readmemh("memfile.dat",RAM);
end

assign rd = RAM[a[31:2]]; // word aligned
always @(posedge clk)
if (we)
if(sbyte)
        RAM[a[31:2]] <= {RAM[a[31:2]][31:8], wd[7:0]};
else
        RAM[a[31:2]] <= wd;
endmodule
```

A.9 mips.sv

```
// pipelined MIPS processor
module mips(input logic clk, reset,
        output logic [31:0] pcF,
        input logic [31:0] instrF,
        output logic memwriteM, sbM,
        output logic [31:0] aluoutM, writedataM,
        input logic [31:0] readdataM);
logic [5:0] opD, functD;
logic [1:0] regdstE;
logic alusrcE, pcsrcD, memtoregE, memtoregM, memtoregW,
      regwriteE, regwriteM, regwriteW;
logic [3:0] alucontrolE;
logic flushE, equalD, jrD, jalD;
logic [1:0] mfhlW;

controller c(clk, reset, opD, functD, flushE,
equalD,memtoregE, memtoregM,
memtoregW, memwriteM, pcsrcD,
branchD, bneD, alusrcE, regdstE, regwriteE,
regwriteM, regwriteW, jumpD, jalD, jalW, jrD, lbW, sbM,
multordivE, hlwriteE, hlwriteM, hlwriteW, mfhlW,
alucontrolE
);

datapath dp(clk, reset, memtoregE, memtoregM, memtoregW,
       pcsrcD, branchD, bneD,
alusrcE, regdstE, regwriteE, regwriteM, regwriteW, jumpD
      , jalD, jalW, jrD, lbW,
multordivE, hlwriteE, hlwriteM, hlwriteW, mfhlW,
      alucontrolE, equalD, pcF, instrF,
aluoutM, writedataM, readdataM, opD, functD, flushE
);
endmodule
```

A.10 topmulti.sv

```
//------------------------------------------------
// topmulti.sv
// David_Harris@hmc.edu 9 November 2005
// Update to SystemVerilog 17 Nov 2010 DMH
// Top level system including multicycle MIPS
// and unified memory
//------------------------------------------------

module top(input logic clk, reset,
        output logic [31:0] writedata, dataadr,
        output logic memwrite);

logic [31:0] pc, instr, readdata;
logic sb;

// instantiate processor and memories
mips mips(clk, reset, pc, instr, memwrite, sb, dataadr,
      writedata, readdata);

imem imem(pc[7:2], instr);

dmem dmem(clk, memwrite, sb, dataadr, writedata,
      readdata);
endmodule
```

A.11 hazard.sv

```
module hazard(input logic [4:0] rsD, rtD, rsE, rtE,
        input logic [4:0] writeregE, writeregM,
              writeregW,
        input logic regwriteE, regwriteM, regwriteW,
        input logic memtoregE, memtoregM, branchD, bneD,
        output logic forwardaD, forwardbD,
        output logic [1:0] forwardaE, forwardbE,
        output logic stallF, stallD,
        flushE);
```

```
logic lwstallD, branchstallD;

// forwarding sources to D stage (branch equality)
assign forwardaD = (rsD !=0 & rsD == writeregM &
      regwriteM);
assign forwardbD = (rtD !=0 & rtD == writeregM &
      regwriteM);

// forwarding sources to E stage (ALU)
always_comb
begin
forwardaE = 2'b00; forwardbE = 2'b00;
        if (rsE != 0)
                if (rsE == writeregM & regwriteM)
                                forwardaE = 2'b10;

                else if (rsE == writeregW & regwriteW)
                                forwardaE = 2'b01;

        if (rtE != 0)
                if (rtE == writeregM & regwriteM)
                     forwardbE = 2'b10;

                else if (rtE == writeregW & regwriteW)
                     forwardbE = 2'b01;
end

// stalls
assign #1 lwstallD = memtoregE & (rtE == rsD | rtE ==
      rtD);
assign #1 branchstallD = (branchD | bneD) & (regwriteE &
       (writeregE == rsD | writeregE == rtD) |
                        memtoregM &(writeregM == rsD |
                                writeregM == rtD));
assign #1 stallD = lwstallD | branchstallD;
assign #1 stallF = stallD;

// stalling D stalls all previous stages
assign #1 flushE = stallD;

// stalling D flushes next stage
// Note: not necessary to stall D stage on store
// if source comes from load;
// instead, another bypass network could
// be added from W to M
endmodule
```

A.12 testbench.sv

```
module testbench();
logic clk;
logic reset;
logic [31:0] writedata, dataadr;
logic memwrite;

// instantiate device to be tested
top dut(clk, reset, writedata, dataadr, memwrite);

// initialize test
initial
begin
        reset <= 1; # 22; reset <= 0;
end

// generate clock to sequence tests
always
begin
        clk <= 1; # 5; clk <= 0; # 5;
end

// check results
always @(negedge clk)
begin
        if(memwrite) begin
        if(dataadr === 84 & writedata === 7) begin
                $display("Simulation succeeded");
                $stop;
        end else if (dataadr !== 80) begin
                $display("Simulation failed");
                $stop;
        end
        end
end
endmodule
```

B Test Programs

B.1 Textbook Test Program

stores 7 in memory location 84.

```
main:  addi $2, $0, 5
addi $3, $0, 12
addi $7, $3, 9
or $4, $7, $2
and $5, $3, $4
add $5, $5, $4
beq $5, $7, end
slt $4, $3, $4
beq $4, $0, around
addi $5, $0, 0
around:  slt $4, $7, $2
add $7, $4, $5
sub $7, $7, $2
sw $7, 68($3)
lw $2, 80($0)
j end
addi $2, $0, 1
end:  sw $2, 84($0)
```

B.2 Series Sum

Adds the numbers from 1 to 10 inclusive.

```
main:  addi $7, $0, 10
add $a0, $0, $7
jal seriesSum
```

```
sw $v0, 84($0)
seriesSum:
add $v0, $0, $0
loop:
add $v0, $v0, $a0
addi $a0, $a0, -1
bne $0, $a0, loop
jr $ra
```

B.3 Recursive Factorial

calculates the factorial of a number, in this test the number is 5.

```
addi $sp, $0, 400
main:  addi $a0, $0, 5
jal factorial
add $s0, $v0, $0
sw $s0, 84($0)
factorial:  addi $sp, $sp,
-8
sw $a0, 4($sp)
sw $ra, 0($sp)
addi $t0, $0, 2
slt $t0, $a0, $t0 beq $t0,
$0, else
addi $v0, $0, 1
addi $sp, $sp, 8
jr $ra
else:  addi $a0, $a0, -1
jal factorial
lw $ra, 0($sp)
lw $a0, 4($sp)
addi $sp, $sp, 8
mult $a0, $v0
mflo $v0
jr $ra
```