Certificate Of Originality

This is to certify that the project work titled **Optimized MIPS** being submitted by **Gaurav Singhal, Mayank Kaushik and Anvay Virkar** as a part of the course VLSI-I, is a record of bonafide work carried out by them under my guidance and supervision at the **Department of Electrical and Computer Engineering , University of Texas at Austin**.

Dr. Adnan Aziz
Department of Electrical and Computer Engineering,
University of Texas at Austin

Contents

1 Baseline MIPS architecture

The MIPS32 architecture is a simple 32-bit RISC architecture "with relatively few idiosyncracies". The baseline architecture uses 32-bit instruction encodings but only eight 8-bit *General Purpose Registers* which are written as $0 to $7. The Program Counter PC is also 8-bit. Register $0 is hardwired to 0.

1.1 Baseline MIPS ISA

The baseline architecture supports the instructions shown in the table below:

Instruction	**Function**	**Enc.**	**op**	**funct**
`add $rd, $rs1, $rs2`	addition: `rd <- rs1 + rs2`	R	000000	100000
`sub $rd, $rs1, $rs2`	subtraction: `rd <- rs1 - rs2`	R	000000	100010
`and $rd, $rs1, $rs2`	bitwise and: `rd <- rs1 & rs2`	R	000000	100100
`or  $rd, $rs1, $rs2`	bitwise or: `rd <- rs1 | rs2`	R	000000	100101

`slt $rd, $rs1, $rs2`	set less than:
	`rd <- 1 if rs1 < rs2`
	`rd <- 0 otherwise`
		R	000000	101010

`addi $rd, $rs1, imm`	add immediate: `rd <- rs1 + imm`	I	001000	n/a
`beq  $rd, $rs1, imm`	branch if equal: `PC <- PC + imm`	I	000100	n/a
`j destination`	jump: `PC <- destination`	J	000010	n/a
`lb $rd, imm($rs1)`	load byte: `rd <- mem[rs + imm]`	I	100000	n/a
`sb $rd, imm($rs1)`	store byte: `mem[rs1 + imm] <- rs1`	I	101000	n/a

Figure 1: Baseline MIPS Instruction Set Architecture

Each instruction is encoded in one of the three templates: R, I and J. The high 6 bits of the formats specify the opcode of the instruction.

- **R-type (Register-based):**
  This instruction template is used for arithemtic and logical

instructions. It specifies two source registers and one destination register. R-type instructions have their high 6-bits as 0; they use additional 6 function bits to specify the ALU operation.

- **I-type (Immediate):**
  A 16-bit constant embedded into the instruction is referred to as an *immediate*. I-type instructions specify the destination register, one source register and one immediate. They are used to add immediate values to register contents, or to add immediate address offsets to a base value in a register.

- **J-type (Jump):**
  J-type instructions contain in them the opcode and the absolute address of the destination.

1.2 Multicycle MIPS Microarchitecture

The baseline architecture is the multicycle MIPS microarchitecture. This is shown in Fig. 2. Following standard conventions, rounded rectangles are multiplexers, the ovals are control logic.

Instruction execution generally flows from left to right. The Program Counter (PC) specifies the address of the instruction. The instruction is loaded one byte at a time over four cycles from an off-chip memory into the 32-bit instruction register. The opcode is sent to the controller, which sequences the datapath through the correct operations to execute the instruction. The result of an (ALU) operation is captured in the ALUout register.

The controller is a finite state machine that generates multiplexer select signals and register enables to sequence the datapath. The controller produces a 2-bit `aluop`. The ALU Control

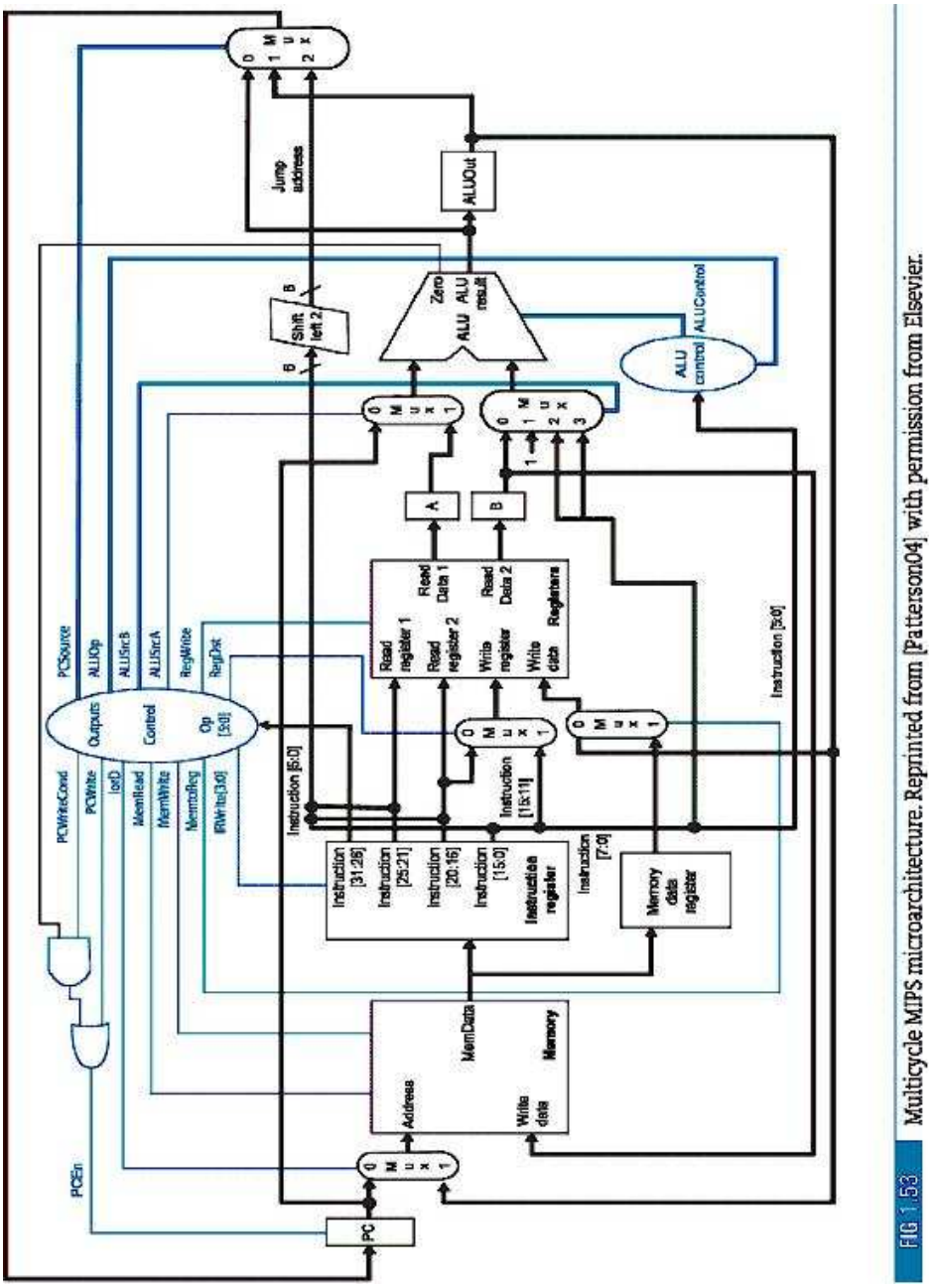

Figure 2: Baseline MIPS Implementation

aluop	funct	alucontrol	Meaning
00	x	010	ADD
01	x	110	SUB
10	100000	010	ADD
10	100010	110	SUB
10	100100	000	AND
10	100101	001	OR
10	101010	111	SLT
11	x	x	undefined

Figure 3: ALU Control Determination

unit uses combinational logic to compute a 3-bit `alucontrol` signal from the `aluop` and `funct` fields as shown in Fig. 3.

2 Enhanced MIPS: A Brief Overview

Following is a list of enhancements made to the baseline architecture:

- **Larger Register File** The baseline architecture has 8 registers, names $0 through $7, each of 32-bits. The new architecture has 32 registers, each register being 32-bits in size, denoted as $0 though $31. Register 0 always has the value 0. A larger register file increases the performance of the processor. A very small number of registers however, necessitates the re-use of registers and which becomes the bottleneck in pipelining — an issue addressed more deeply in Section 3.

- **Separate Instruction and Data Memories (*Harvard Architecture*)** The baseline architecture makes use of a *unified* memory, wherein the instructions and data are both stored in the same memory. This gives rise to *structural hazards* the details of which are postponed until Section 4.

- **Single-cycle instruction fetch** The baseline architecture simulates an 8-bit wide memory from which the MIPS can read/write data 1-byte per cycle. However, that approach is too antiquated and impractical. Thanks to *interleaving* (and doubtlessly, *caching*), the fetch time can be reduced considerably. The Enhanced MIPS employs this approach and therefore, instruction fetch occurs in a single cycle. Also, the PC was made 32-bits.

- **Pipelining** The advantage of the hardware in the datapath is that all the functional units therein can execute in parallel. This means that it is possible to have multiple instruc-

tions being operated upon concurrently, each instruction being in a different *phase* of operation. This technique of overlapping instruction execution is called *Pipelining*. All the processors today are pipelined.

- **Data-forwarding** Simultaneous execution of instructions gives rise to *Flow Dependencies*. A very common technique of preventing the performance being affected by dependencies is by Data-Forwarding. Details of the concept and implementation of this enhancement are spared until Section 4.2.1

- **Branch Prediction: BTFN** BTFN stands for "Backward Taken – Forward Not Taken" Branch Prediction. It is a *static* branch prediction scheme. Branch Predictions help to improve the efficiency of the pipeline.

3 The Enhancement: Pipelining

Pipelining is a performance-improving technique whereby the operations of a number of instructions are made to overlap in time. At a given instant, each instruction is in a different *phase* or a step of the computation. Each of these steps is called a *pipe stage* or a *pipe segment*. The performance of a pipeline is measured in terms of the parameter *IPC* or "Instructions Per Cycle". Such a parameter is called a *System Design Metric*.

3.1 Non-pipelined Baseline Architecture

The baseline architecture was a *multi-cycle* implementation. Every instruction in the instruction set could be implemented in at most 5 clock cycles. The operations taking place in each cycle are given below:

1. **Instruction Fetch**
   (a) Send the address in the PC to memory.
   (b) Fetch the instruction from memory.
   (c) Update the PC to point to the next instruction.
2. **Instruction Decode**
   (a) Decode the instruction.
   (b) Read the operands of registers.
3. **Execution/Address Computation** Depending upon the type of instruction, the ALU performs one of the following operations:

(a) Register-Register Operation: The arithmetical/logical operation is performed on the operands found in the Instruction Decode cycle. The

(b) Register-Immediate Operation: The arithmetical/logical operation is performed on the first source register and a signed offset (extended to 32 bits).

(c) Memory Reference: An effective address is calculated by adding a signed offset to a base register.

4. **Memory Access**

   (a) For a load instruction, data is read from the memory location whose address was generated in the previous cycle.

   (b) For a store instruction, the data is written into the memory location whose address was found in the previous cycle.

5. **Write-back** In this cycle, the destination register is loaded with the data fetched (for a load instruction) or the ALU result (for a Register-Register ALU operation).

3.2 The Classical RISC Pipeline

While conceiving a pipelined version for an initially non-pipelined machine, the most essential thing to do is to identify those components of the datapath that can operate in parallel. In effect, the datapath is being divided into stages. Each stage takes one clock cycle to execute. Notice how conducive the baseline architecture was to pipelining. To implement the pipeline, it is necessary to make the following modifications:

	Clock number								
Instruction Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$	IF	ID	EX	MEM	WB				
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Figure 4: Execution Pattern for a 5-stage Classical Pipeline

- Add latches between two pipeline stages. These are called *pipeline registers* or *pipeline latches*. Each stage of the pipeline operates on the data in its latches and as an output, updates the registers of the next stage, at the end of the cycle. The latches ensure that each stage is maintained in its own phase of operation; each stage supports an independent operation (which flows down the pipe).
- Each new clock cycle initiates a new instruction cycle. An *instruction cycle* is the time required to complete one instruction. In the MIPS, an instruction cycle is made of five clock cycles.

We now have five pipeline stages and four pipeline registers. The *exeuction pattern* of a pipelined machine is shown in Fig. 4. The operations taking place in the pipeline stages are described below:

1. **Instruction Fetch Stage (IF):**

```
 IR←MEM[PC]
NPC←PC+4
```

   Operation The address in the PC is sent out to the instruction memory and the instruction is fetched. The instruction is held in the Instruction Register (IR). The

PC is incremented by 4, a word being 4 bytes. Likewise, the next value of the PC (denoted NPC) is latched into the pipeline latches.

2. **Instruction Decode/Register Fetch Stage (ID):**

```
  A←REG[RS1]           ; Source Reg. #1
  B←REG[RS2]           ; Source Reg. #2
Imm←SEXT(IR[15:0])     ; Sign-extended Immediate
```

Operation The instruction held in IR is decoded. The register specifiers in the instruction fields (denoted `RS1` and `RS2` in the above pseudo-code) are used to read the operands from the register file and copied into the pipeline registers. A and B are a part of the pipeline registers. The Immediate field, the lower half-word of the IR, is sign-extended and stored in the pipeline latches.

Decoding and Operand-read are two parallel operations. This is possible because the opcode and the register specifier bits always occur at a fixed position in a given instruction word. This technique, characteristic of RISC designs, is called *uniform decoding* or *fixed-field encoding*.

3. **Exeution/Address Computation Stage:**
The operation taking place in this stage depends upon the type of instruction. There are four cases to consider.

(a) **Memory Reference:**

```
ALU_out←A + Imm
```

Operation The sign-extended immediate is added to the first operand and stores the address in the pipeline

register.

(b) **Register-Register Operation:**

```
ALU_out←A op B
```

(c) **Register-Immediate Operation:**

```
ALU_out←A op Imm
```

(d) **Branches:**

```
Address←NPC + (Imm << 2)
   Cond←A==B? 1: 0
```

4. **Memory Access/Branch Completion Stage (MEM)**
   There are two cases to consider.

   (a) **Memory Reference**

   ```
   PC←NPC
   if (LOAD)  MDR←MEM[Address]
   if (STORE) MEM[Address←B
   ```

   Operation If the instruction is a load, read the data and place it in the MDR register. If it is a store, then write the data to the Data Memory. In either case, it is the Address computed in the previous state that is used to access memory.

   (b) **Branch**

   ```
   if (cond) PC←Address
   ```

5. **Write Back Stage (WB)**

   (a) **Register-Register Operation**

   ```
   REG[RD1]←ALU_out
   ```

   (b) **Register-Immediate Operation**

   ```
   REG[RD2]←ALU_out
   ```

(c) **Load instruction**
`REG[RD2]←MDR`

Operation Write the data (from MDR or ALU) to the destination register.

The modified datapath is shown in Fig. **??**. Note how the pipeline registers have been named. A pipeline register sitting between stages 'X' and 'Y' is named 'X/Y'.

3.2.1 Pipeline Register Implementation

The concept of having Pipeline Registers between two stages seems to be quite logical and therefore, trivial at first glance. However, there are certain critical design issues about them which are necessary to take into account. Prior to that, it is worthwhile reviewing the use of the pipeline registers:

- The components in the datapath need to be grouped to facilitate pipelining. The grouping is achieved by means of pipelining registers, which serve to delimit the stages of the pipeline.
- Pipeline registers ensure that the different stages of the pipeline do not interfere with each other.
- Pipeline registers carry data and control signals from one stage to another.
- Pipeline registers ensure *data integrity* by responding to the changed signals only at one instant, namely, the active edge of the clock.

A pipeline is an *operation sequencer*. A pipeline initiates a sequence of dataflow. Every stage in the pipeline need to abide the dataflow protocol. Naturally, there is the need for a sequencing mechanism. This mechanism in built into the system due to the pipeline registers. There are three general sequencing methods:

1. Flip-flops
2. Two-phase latches
3. Pulsed latches

Pulsed latches have hold time risk. Therefore, they have not been used. The advantages of Two-phase latches is that they have high skew tolerance and allow for substantial time borrowing. However, flip-flops are the easiest to use and are supported by all tools. Therefore, flip-flops have been used to implement pipeline registers.

3.2.2 Performance of the Basic Pipeline

Earlier, the philosophy behind the increase in performance due to pipelining was put forth qualitatively. Quantitavely, a measure of performance is the *speedup* which is defined, in general, for a pipeline as:

$$\text{Speedup from Pipelining} = \frac{\text{Average CPI for unpipelined}}{\text{Average CPI for pipelined}}$$

where CPI stands for *Clock cycles per instructions* which is also a system design metric.

It may be noteworthy to point out the fact that pipelining improves the throughput of the system. However, it does not

reduce the instruction cycle time. Thus, the program as a whole runs faster, even though no single instruction has been made fast. In fact, the instruction cycle time increases by a small amount due to control overhead of the pipeline. Overhead is a result of pipeline register delay and clock skew. The clock skew imposes a lower limit on the clock cycle. The clock can be no less than the sum of the clock skew and latch overhead.

A limitation of the pipelining is *Pipeline imbalance*. The pipeline can be as fast as its slowest stage.[1] It is possible that a stage may be slower than the others due to *transistor assymtry* effects.

However, the immediate countereffects of a pipeline that directly affect performance are *pipeline hazards*.

[1] A chain is only as strong as its weakest link.

4 Various Pipeline Hazards And Solutions

Hazards are situations in which the sequential flow of operations through the pipeline gets stunted, and the next instruction which was expected to execute has to wait until the block in the pipeline — or a *pipeline bubble* — is cleared. There are three types of stalls:

- Structural Hazards
- Data Hazards
- Control Hazards

This section is devoted to the program conditions that cause hazards and design strategies to work around them.

A hazard causes an instruction to *stall*. All instructions (i.e., those at the *front-end*) issued later than that instruction are also stalled. Instructions issued earlier (i.e., those in the *back-end*) continue; they need to continue for the hazard to be cleared.

4.1 Structural Hazards

Structural hazards are *resource conflicts*. A conflict arises when two stages in the pipeline attempt to access the same resource (that cannot service more than one client). For example, in Fig. 4, in clock cycle #4, both MEM and IF stages attempt to access memory. In a *Von Neumann* architecture, which has a unified memory, having both instructions and data in the same memory, this leads to a conflict.
Solution: Separate the instruction memory and data memory. Our design conforms to this policy, or Harvard architecture, as mentioned earlier.

Instruction	Clock number					
	1	2	3	4	5	6
`ADD $1, $2, $3`	IF	ID	EX	MEM	WB ⋆	
`ADD $4, $5, $1`		IF	ID†	EX	MEM	WB

⋆: Producer
†: Consumer

Figure 5: Data Hazard Execution Pattern

4.2 Data Hazards

Consider a code snippet shown below:

```
ADD $1, $2, $3
ADD $4, $5, $1
```

Now consider the execution pattern of this code as shown in Fig. 5. The first instruction writes into $1 in the fourth stage. But the second instruction needs the value in the second stage, early in time! This leads to a data hazard. A data hazard has been caused as a result of a *flow dependency*. The second ADD should not be allowed to continue until the first has gone past the MEM stage. This is done by inserting bubbles into the pipeline, essentially by making the pipeline registers as 0. Making the pipeline registers 0 has the effect of making the IR in them 0, which is nothing but the instruction `ADD $0, $0, $0`. This instruction is effectively a `NOP` as register 0 is hardwired to zero.

One way out: Stall the pipeline.

A better way out: *Data forwarding.*

4.2.1 Data Forwarding and Interlocks

Data Forwarding is also called *bypassing* and *short-circuiting*. Data forwarding is based on the understanding that the result is not needed by the second ADD until the first ADD produces it. If the result can be moved from the pipeline register where the first ADD stores it, then the need for a stall can be avoided. Using this observation, data forwarding works as follows:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.

2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read in the register file.

From Fig. 4, we can see that data needs to be forwarded from an instruction that started 2 cycles earlier.

Note that data forwarding does not necessarily eliminate stalls. For example, consider the following code:

```
LD   R1, 0(R2)
DSUB R4, R1, R5
AND  R6, R1, R7
OR   R8, R1, R9
```

The execution pattern for this code is shown in Fig. 6 Notice that the `LD` instruction gives the data at the end of cycle 4, but `DSUB` needs it at the beginning of cycle 4. Thus the forwarded result arrives too late — at the end of a clock cycle, when it is needed at the beginning.

Instruction	Clock number					
	1	2	3	4	5	6
`LD R1, 0(R2)`	IF	ID	EX	MEM⋆	WB	
`DSUB R4, R1, R5`		IF	ID	EX†	MEM	WB

⋆: Producer: Value produced at end of cycle
†: Consumer: Value required at start of cycle

Figure 6: Need of stall inspite of Forwarding

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
`LD R1, 0(R2)`	IF	ID	EX	MEM	WB				
`DSUB R4, R1, R5`		IF	ID	stall	EX	MEM	WB		
`AND R6, R1, R7`			IF	stall	ID	EX	MEM	WB	
`OR R8, R1, R9`				stall	IF	ID	EX	MEM	WB

Figure 7: Data Forwarding with stalls

The load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, it is necessary to freeze the pipeline (stages prior to the one causing a stall) or in other words, to add a hardware *pipeline interlock* to correctly insert a bubble until the stall is cleared. The CPI for the stalled instruction increases by the length of the stall.

The correct execution pattern for the code is shown in Fig. 7

4.2.2 Implementation of Control

The process of letting an instruction move from the instruction decode stage (ID) to the execution stage (EX) is called *issue* of an instruction. For the MIPS pipeline, all data hazards are checked during the ID phase of the pipeline. If a data hazard exists, the instruction is stalled before it is issued. Likewise, we

Opcode in ID/EX	Opcode in IF/ID	Interlock test
Load	Register-register ALU	`ID/EX.IR[RS2]==IF/ID.IR[RS1]`
Load	Register-register ALU	`ID/EX.IR[RS2]==IF/ID.IR[RS2]`
Load	Load, Store, ALU Imm, Branch	`ID/EX.IR[RS2]==IF/ID.IR[RS1]`

Figure 8: Load Interlock Logic

can determine the forwarding that will be needed during ID ans set the appropriate controls then. The implentation of a load interlock check can be implemented in ID and the implementation of the forwarding paths to the ALU inputs during EX.

Load Interlock Logic:

If there is a data hazard with the source instruction being a load, the load instruction will be in EX stage when an instruction that needs the data will be in the ID stage. All possible hazard conditions are listed in the Fig. 8. Once a hazard has been detected, the control unit must insert the pipeline stall and prevent the instructions in the IF and ID stages from advancing. All control information is carried in the pipeline registers. Thus, when we detect a hazard, we need only change the control portion of the ID/EX pipeline register to all 0s. The all zero opcode acts as a `NOP` as it is actually `DADD R0, R0, R0`. In addition, we simply re-circulate the contents of the IF/ID registers to hold the stalled instruction.

Forwarding Logic:

Implementing the forwarding logic is similar, except that there are more cases to consider. We observe that the pipeline registers contain both the data to be forwarded as well as the source and destination register fields. Thus, we can implement the forwarding by a comparison of the destination registers of

the IR of EX/MEM and MEM/WB stages against the source registers in IR of ID/EX and EX/MEM. All these cases are shown in Fig. 9. In hardware, the table in the figure translates to comparators and combinational logic that determine when a forwarding path needs to be enabled. We added additional multiplexers at the ALU inputs and add connections from the pipeline registers that are used to forward the results.

4.3 Control Hazards

Control hazards are hazards arising due to control instructions like branch. Control instructions are those instructions that change the PC. Branches cause stalls due to the uncertainty of the outcome of the comparison operation. The simplest solution of dealing with control hazards is to freeze or flush the pipeline holding or deleting any instructions after the branch. It is possible to complete the decision of the comparison by the end of the ID cycle by adding a comparator in the ID stage. To take advantage of the early decision, both PCs (those of the branch and of the fall-through) must be computed early. Computing the branch target address requires an additional adder, since the ALU is not usable until EX stage. This causes only 1-cycle stall on branches.

For source instruction in: EX/MEM			
Source Op-code	**Destination Op-code**	**Dest. of for-warded result**	**Test**
Reg-Reg	Reg-Reg, Immediate, Load, Store, Branch	A	EX/MEM.IR[RD] == ID/EX.IR[RS1]
Reg-Reg	Reg-Reg	B	EX/MEM.IR[RD] == ID/EX.IR[RS2]
Imm	Reg-Reg, Imm, Load, Store, Branch	A	EX/MEM.IR[RS2] == ID/EX.IR[RS1]
Imm	Reg-Reg	B	EX/MEM.IR[RS2] == ID/EX.IR[RS2]
For source instruction in: MEM/WB			
Source Op-code	**Destination Op-code**	**Dest. of for-warded result**	**Test**
Reg-Reg	Reg-Reg, Imm, Load, Store, Branch	A	MEM/WB.IR[RD] == ID/EX.IR[RS1]
Reg-Reg	Reg-Reg	B	MEM/WB.IR[RD] == ID/EX.IR[RS2]
Imm	Reg-Reg, Imm, Load, Store, Branch	A	MEM/WB.IR[RS2] == ID/EX.IR[RS1]
Imm	Reg-Reg	B	MEM/WB.IR[RS2] == ID/EX.IR[RS2]
Load	Reg-Reg, Imm, Load, Store, Branch	A	MEM/WB.IR[RS2] == ID/EX.IR[RS1]
Load	Reg-Reg	B	MEM/WB.IR[RS2] == ID/EX.IR[Rs2]

Figure 9: Data Forwarding Logic

5 Other Enhancements: Branch Prediction

Branch Prediction mechanisms are *speculative execution* techniques. The purpose of branch prediction is to reduce the stall period of the pipeline. In order to reduce the stall, a prediction is made as to whether the branch will be taken or not. If the prediction is correct, then no stalling of the pipeline will be necessary. However, if the prediction turns out to be incorrect, there is a latency referred to as *Branch Misprediction penalty.* Branch Prediction mechanisms aim to reduce the misprediction penalty.

There are two approaches to Branch Prediction:

- Static Branch Prediction
- Dynamic Branch Prediction

Static branch prediction techniques are those that make a prediction about the branch at compile-time. Dynamic branch predictors make a comparison at run-time. Implementation of dynamic branch prediction increases significantly the area and the power consumption of the chip. We have implemented one of the best known static prediction scheme called *BTFN—Backward Taken–Forward Not taken.*

The BTFN mechanism predicts that all backward branches will be taken, and all forward branches will not be taken. This is due to the observation that all loops have much more backward branches than forward braches. For example, consider the following loop:

```
BACK: LD R1, #COUNT  ; Counter
      . . .
      . . .
```

```
SLT R2, R1, R0   ; If count==0, R2=1
BEQ R2, R0, BACK ; If count<>0, branch back
. . .
```

If the counter value is set to n, then the backward branch is taken $(n-1)$ times and the forward path is taken 1 time. The BTFN scheme predicts each of the n times that it will be taken. Thus out of n, there is only 1 misprediction. Thus the accuracy of the BTFN scheme is $\frac{n-1}{n}$ which is very high.

The final enhanced MIPS architecture Implementation with Pipelining, Data Forwarding and Branch Prediction is shown in Fig. 10.

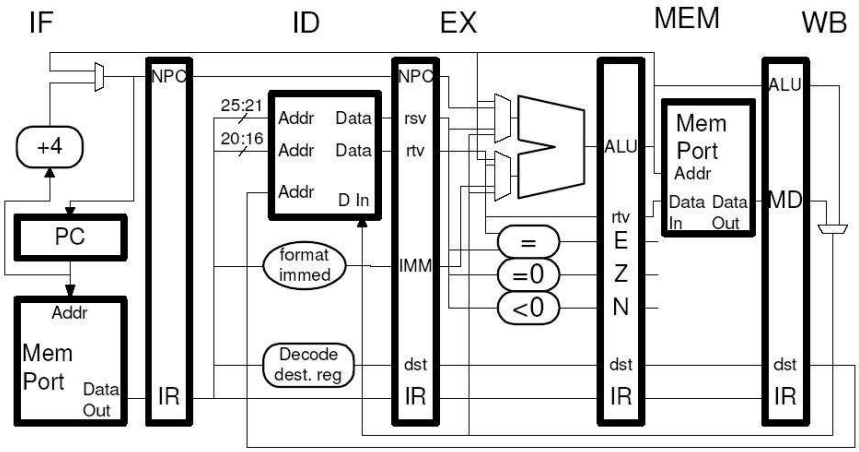

Figure 10: Enhanced MIPS Implementation

6 Optimzied Adder Design

The delay encountered in a microprocessor's ALU is a major factor influencing the maximum frequency that the chip can operate at. The delay encountered in the simple ALU in our implementation of an optimized MIPS microprocessor depends a great deal on the adder that makes up the ALU, and also the register file. To further optimize the design and increase the frequency at which the chip can be operated, both the register file and the adder will have to be optimized.

The following is a discussion of the delays encountered in various VLSI adder implementations. A comparison of the delays encountered in both static and dynamic implementations of various popular adder topologies is presented, drawn from various published works.

In the course of VLSI processor design it is very important to choose the adder topology that would yield the desired performance. The speed of a VLSI adder depends on many factors: the technology of implementation (and its own internal rules), circuit family used for the implementation, sizing of transistors, chosen topology of the VLSI adder, and many other second order effect parameters. Knowles [4] has shown how different topologies may influence fan-out and wiring density thus influencing design decisions and yielding better area/power than known cases.

Logical Effort (LE) can be used to estimate the speed of various VLSI adders [1]. Logical Effort methodology takes into account the fact that the speed of a digital circuit block is dependent on its output load (fan-out) and its topology (fan-in). Further, LE introduces technology independence by normalizing the speed to that of a minimal size inverter which makes

the comparisons of different topologies, implemented in different technologies, possible.

The following are the results of performing critical path analysis using Logical Effort technique to compare performance, useing several representative topologies, as published in [1]. The adders that were examined were:

1. Static:
   (a) Kogge-Stone (KS) radix-2 [5]
   (b) Mux-based carry-select [2]
   (c) Han-Carlson (HC) radix-2 [3, 7]
2. Dynamic:
   (a) KS radix-2, Ling Adder [6]
   (b) HC radix-2
   (c) CLA adder with 4-bit grouping.

The results obtained using Logical Effort were compared with the results obtained using H-SPICE simulation. The comparison results are shown in Fig. 11.

Wire delays were accounted for by estimating the length of the wire and assigning appropriate delay to it, however, the portion of the wire delay was not significant (less than 10% of the total delay) due to the proximity of the cells.

The obvious observation is that there is a huge difference between Static CMOS and Dynamic CMOS implementations. This demonstrates the dependency on logic design style, and thus static CMOS adders are rarely seen in places where high speed is required. Instead dynamic implementations like domino logic are used.

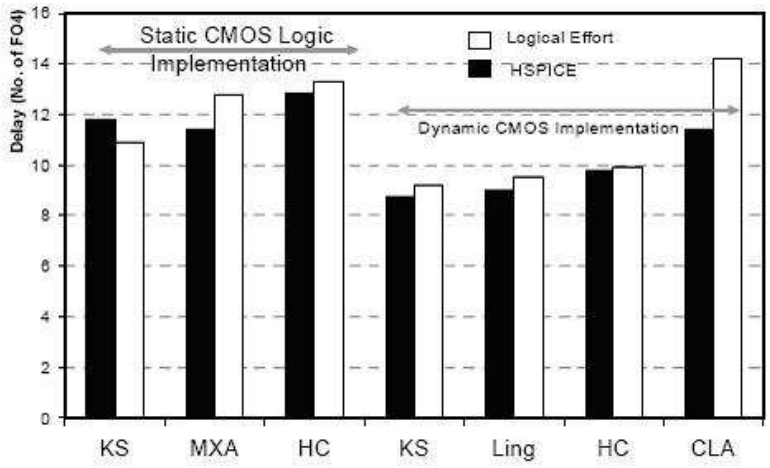

Figure 11: Speed estimation of various VLSI adders using Logical Effort vs. H-SPICE results

A more rigorous quantitative estimation and evaluation of different VLSI adder topologies has been done in [8], which is cited here. Before the analysis, it is necessary to characterize the technology used. This step needs to be done only once, but it improves the accuracy of the LE since the characteristics of the technology are taken into account. Characterization is performed using SPICE simulation of the gate delay for various output loads driving a copy of itself, according to the LE rules. This is repeated for each cell used in the logic library. Characterization of dynamic gates requires special attention due to the fact that only one transition is of interest. Obtained results are compared to that of an inverter and parameters such as parasitic delay (p) and effort (g) were normalized with respect to that of an inverter. Select results are shown in Fig. 6 below:

This step preserves LE features, allowing delay results to be presented in terms of fan-out of 4 (FO4) delay, relatively in-

0.10μm technology, FO4=19pS

Gate type	LE (g)	Parasitics (p inv)
Inverter	1	1
Dyn. Nand	0.6	1.34
Dyn. CM	0.6	1.62
Dyn. CM-4N	1	3.71
Static CM	1.48	2.53
Mux	1.68	2.93
XOR	1.69	2.97

CM: Carry-Merge cell
Dyn: Dynamic

Figure 12: Normalized LE parameters

dependent of the technology of implementation. The LE-based delay estimation tool works on the logic stages in the critical path, assigning branch effort (bi), logical effort (gi) and parasitic effort (pi) to each gate (Fig. 13).

In computing the branch effort, we take into consideration the worst-case interconnect at each stage. In a 64-bit Kogge-Stone adder, the worst-case interconnect in stage 6 (CM C30) spans 32 bit-slices. We make an assumption that the adder bit-pitch is 10um, which would result in a 320μm wire. To account for the propagation delay through a wire we incorporate an Elmore delay model in Fig. 13, which corresponds to the critical-path interconnect delay in the adder.

A comparison of representative VLSI adders implemented in static and dynamic CMOS design style is presented in Fig. 14. It is interesting to see that there are indeed very small speed differences between the three fastest dynamic adders: KS, HC and Quarternary (QT).

Prefix-2 Kogge-Stone (Static)

Stages	Bit Span	Branch Effort (b_i)	LE (g_i)	Parasitic (p_i)	Total Branch (B)	Total LE (G)	Path Effort (F)	Fopt (f)	Effort Delay (ps)	Parasitic Delay (ps)	Wire Delay (ps)	Total Delay (ps)	Total Delay (FO4)
g0 (NAND2)	0	2.0	1.11	1.84									
C0 (OAI)	2	2.2	1.55	2.26									
C2 (AOI)	4	2.4	1.52	2.76									
C6 (OAI)	8	2.8	1.55	2.26									
C14 (AOI)	16	3.6	1.52	2.76	1.66E+03	2.26E+01	3.76E+04	3.22	106	88	14	**209**	**11.0**
C30 (OAI)	32	5.2	1.55	2.26									
C62 (AOI)	0	1.0	1.52	2.76									
S63 (TGXORs)	0	1.0	1.56	2.59									
INV (INV)	0	3.0	1.00	1.00									

Prefix-2 Kogge-Stone (Dynamic)

Stages	Bit Span	Branch Effort (b_i)	LE (g_i)	Parasitic (p_i)	Total Branch (B)	Total LE (G)	Path Effort (F)	Fopt (f)	Effort Delay (ps)	Parasitic Delay (ps)	Wire Delay (ps)	Total Delay (ps)	Total Delay (FO4)
g0 (Dk1ND2)	0	2.0	1.02	1.34									
C0 (OAI)	2	2.2	1.36	1.69									
C2 (DAOI)	4	2.4	0.68	1.33									
C6 (OAI)	8	2.8	1.36	1.69									
C14 (DAOI)	16	3.6	0.68	1.33	1.66E+03	1.26E+00	2.09E+03	2.34	77	60	14	**151**	**8.0**
C30 (OAI)	32	5.2	1.36	1.69									
C62 (DAOI)	0	1.0	0.68	1.33									
S63 (TGXORs)	0	1.0	1.56	2.59									
INV (INV)	0	3.0	1.00	1.00									

Figure 13: Delay Comparison of Static and Dynamic implementation of Kogge-Stone Prefix-2 Adder

Adders	Stages	Total Branch (B)	Total LE (G)	Path Effort (F)	f_{opt}	Effort Delay (pS)	Parasitic Delay (pS)	Wire Delay (pS)	Total Delay (pS)	Total Delay (FO4)
Static MXA	15	11600	0.369	4280	1.75	96	93	14	203	10.7
Static KS	9	1660	22.6	37600	3.22	106	88	14	209	11
Static HC	10	1660	22.6	37600	2.87	105	92	14	212	11.1
Dynamic KS	9	1660	1.26	2090	2.34	77	60	14	151	8.0
Dynamic HC	10	1660	1.26	2090	2.15	79	64	14	157	8.26
Dynamic QT	10	1540	2.08	3220	2.24	82	68	8	158	8.3
Dynamic LNG	10	1430	0.973	1400	2.06	76	70	15	161	8.47
Dynamic CLA	14	20600	0.627	12900	1.97	101	81	12	195	10.26

Figure 14: Comparison of representative VLSI adders using Logical Effort (wire delay estimate included)

7 Functional Verification and Testing

Exhaustive testing and verification was carried to check the functional and logical correctness of the implemented enhanced MIPS architecture. First, a test program which tests all the instructions of the ISA was written and was used as a benchmark to check the functional correctness of all the intermediate designs as we went forward towards implementing our final enhanced MIPS.

The assembly code, and hexadecimal encoding of this code is given in Fig. 15.

Program	**Machine Code**
`; Code #1`	
`; Testing Functional Correctness`	
`addi $3, $0, 8`	200300008
`addi $4, $0, -2`	2001fffd
`add  $5, $3, $4`	00642420
`sub  $6, $3, $4`	00643022
`and  $5, $3, $4`	00642424
`or   $6, $3, $4`	00643025
`slt  $5, $3, $4`	0064242a
`beq  $6, $5, 4`	10c50004
`sb   $6, 2($0)`	a0060002
`lb   $5, 2($0)`	80050002
`j    7`	08000007

Figure 15: Functionality Test

Our final as well as the intermediate implementations passed the test correctly *i.e* the execution trace and output was as expected.

Next a test code was made to test Data forwarding. Data and control dependencies were created in this test case, and the

baseline and pipelined MIPS with dataforwarding was tested against it. The Pipelined MIPS without dataforwarding was halted at various positions due to data dependencies, and hence took more cycles to execute the code, whereas MIPS with data forwarding executed the code in lesser number of cycles, showing the effectiveness of data forwarding.

The assembly code, and hexadecimal encoding of the code to test effectiveness of data forwarding is given in Fig. 16.

Program	**Machine Code**
`; Code #2`	
`; Fibonacci Numbers Program`	
`addi $3, $0, 10`	20030008
`addi $4, $0, 1`	20040001
`addi $5. $0, -1`	2005ffff
`loop: beq $3, $0, end`	10600005
`add $4, $4, $5`	00852020
`sub $5, $4, $5`	00852822
`addi $3, $3, -1`	2063ffff
`j loop`	08000003
`sb $4, 255($0)`	000400ff

Figure 16: Fibonacci Numbers Program

Finally, a test code was written to demonstrate the effectiveness of Branch prediction. The final implementation with Branch prediction was compared against various intermediate designs, and the number of cycles taken to execute each code is given in Fig. 18. Clearly a good branch predictor like **BTFN** results in significant increase in throughput of the entire pipeline, hence increaing performance.

The assembly code, and hexadecimal encoding of the final code to test branch prediction is given in Fig. 17.

The waveforms of final code (to test branch prediction) run-

Program	Machine Code
`; Code #3`	
`; Fibonacci Numbers Program: Backward Loop`	
`addi $3, $0, a`	2003000a
`addi $4, $0, 1`	20040001
`addi $5, $0, -1`	2005ffff
`loop: add $4, $4, $5`	00852020
`sub $5, $4, $5`	00852822
`addi $3, $0, -1`	2063ffff
`bne $3, $0, loop`	1460fffc
`sb $4, 255($0)`	a00400ff

Figure 17: Fibonacci Numbers Program With Backword Loop

Design	# cycles for Code in Fig. 17
Baseline MIPS (32-bit) extended	179
Pipelined (No Data Forwarding and no prediction)	109
Pipelined (Data Forwarding, No Prediction)	75
Pipelined (Data Forwarding, Branch Prediction)	51

Figure 18: Results and Comparison

ning on Pipelined MIPS with Data Forwarding but not Branch prediction & Pipelined MIPS with Data Forwarding and Branch prediction are attached as an Appendix to the report.

8 Results and Conclusion

The first change that we made to the basic design in [10] was to extend the fetch bandwidth from 8-bit to 32-bit. This resulted in a single-cycle fetch, which is a basic requirement for designing a pipeline [9].

Next we implemented a 5-stage pipeline. The pipelining of instructions took advantage of parallelism in hardware, thereby resulting in greater throughput.

The addition of a pipeline resulted in various pipeline hazards, viz., Structural, Data and Control Hazards. Structural hazards were avoided by having separate instruction and data caches, and by having a separate adder for incrementing the Program Counter. Data Hazards were removed by using aggressive Data Forwarding. Control Hazards were reduced by using an effective branch prediction mechanism called BTFN.

A comprehensive and exhaustive test plan was developed to check the functionality of the implemented design. First, a test program which tests all the instructions of the ISA was written and was used as a benchmark to check the functional correctness of all the intermediate designs as we went forward towards implementing our final enhanced MIPS. *Our final as well as the intermediate implementations passed the test correctly i.e the execution trace and output was as expected.* Likewise, a test program for verifying Data Forwarding was written. Finally, a test code was made to demonstrate the effectiveness of Branch prediction. The final implementation with Branch prediction was compared against various intermediate designs, and the number of cycles taken to execute each code is given in Fig. 19. Clearly a good branch predictor like **BTFN** results in significant increase in throughput of the entire pipeline, hence

increasing performance.

Design	# cycles for Code in Fig. 17
Baseline MIPS (32-bit) extended	179
Pipelined (No Data Forwarding and no prediction)	109
Pipelined (Data Forwarding, No Prediction)	75
Pipelined (Data Forwarding, Branch Prediction)	51

Figure 19: Results and Comparison

For quantitative comparisons of the various performance parameters of the base-level design as compared to the final pipelined design, the Verilog implementations of both were synthesized and further analyzed using Design Vision.

The Verilog files were read into Design Vision, linked and compiled. The designs were found to be fully synthesizable.

A qualitative discussion of the results obtained is presented below.

8.1 Power analysis

The baseline implementation was reported to have a dynamic power consumption of 6.2159 W, where as the final implementation had a figure of 10.7881 W. The Design Vision reports for power can be found in the Appendix.

The increase in power can be attributed to the increase in the number of functional units. In the analysis reported here, the dynamic power consumption due to switching activity in the values stored in functional units such as the register file is not taken into account by the software. However, even this simple analysis shows the increase in dynamic power dissipation in our final implementation over that in the initial baseline implementation.

8.2 Area analysis

The initial design was reported in Design Vision to have a total area of 235822.875 units, whereas the final design was reported in Design Vision to have a total area of 307056.843750 units. The Design Vision reports for area can be found in the Appendix.

The increase in area can be explained on the basis of the greater number of functional units in the final implementation. In the final pipelined MIPS implementation, an extra adder had to be incorporated so that the fetch stage and the execute stage, both of which need adders, could be carried out in parallel. The adder can be assumed to contribute almost half the total area of 235822.875 units in the initial design. The effect of the extra adder, as well as the contribution of a number of extra functional units added to implement pipelining is evident from the increased area of 307056.843750 units of the final implementation.

8.3 Timing Analysis

An analysis of the timing reports provided by Design Vision for the two implementations shows interesting results. It is found that both the basic implementation and the final pipelined implementation have the same maximum clock frequency for the above values of the areas. The Design Vision reports for timing can be found in the Appendix.

A closer look at the timing reports shows that in the initial implementation, the maximum frequency is constrained by the delay of the adder, which contributes 793.73ps out of the 1940ps of total delay time, where the maximum available time is 2000ps. In contrast, in the final implementation, the maxi-

mum frequency is constrained by the delay of the register file, which contributes 524.17ps out of the 940.78ps of total delay time, where the maximum available time is 1000ps. This difference arises due to the different implementations of the two designs. In the second design, the register file has only a half clock cycle to produce its value, since it is designed to be activated on the negative edge of the clock cycle. This is necessary in order to implement data forwarding while preserving the cycle time.

This leads to the conclusion that in a final optimized version of the pipelined MIPS, both the adder and the register file need to be optimized. The size of conventional register files grow linearly with respect to the number of registers. The increased area contributes to greater delay. Additionally, the increase in the complexity of the decode logic needed to access the various registers in the register file also contributes to the delay. Suggestions for efficient implementation of register files can be found in various published works, most of which suggest hierarchical implementations of the register file. In section 6 on page 24 we have presented comparisions of performance of various adders, taken from various published works.

At first glance, this timing analysis may seem to suggest that no improvement was obtained by pipelining the baseline MIPS, since the maximum achievable clock frequency is the same for both implementations. However, we must remember that the real advantage of pipelining is the increase in throughput, according to the discussion already presented.

APR of the final pipelined implementation was done. Silicon Ensemble Design Summary Report is attached at the end, with the Layout of the chip.

References

[1] H. Dao and V. G. Oklobdzija. Application of logical effort techniques for speed optimization and analysis of representative adders. In *35th Annual Asilomar Conference on Signal, Systems and Computers*, pages 272–279, 2001.

[2] A. Farooqui, V. G. Oklobdzija, and F. Chehrazi. Multiplexer based adder for media signal processing. In *International Symposium on VLSI Technology, Systems and Applications*, 1999.

[3] T. Han, D. A. Carlson, and S. P. Levitan. Vlsi design of high-speed low-area addition circuitry. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 418–422, 1987.

[4] S. Knowles. A family of adders. In *IEEE Symposium on Computer Arithmetic*, pages 30–34, 1999.

[5] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. In *IEEE Transactions on Computers*, pages 786–793, 1973.

[6] H. Ling. High speed binary adder. In *IBM journal of Research and Development*, pages 156–166, 1981.

[7] S. K. Mathew. Sub-500-ps 64-b alus in 0.18m soi/bulk cmos: design and scaling trends. In *IEEE Journal of Solid-State Circuits*, pages 1636–1646, 2001.

[8] V. G. Oklobdzija, B. R. Zeydel, H. Dao, S. Mathew, and R. Krishnamurthy. Energy-delay estimation technique for

high-performance microprocessor vlsi adders. In *IEEE Symposium on Computer Arithmetic*, pages 272–279, 2003.

[9] D. A. Patterson and J. L. Hennessy. *Computer Architecture : A Quantitative Approach.* Morgan Kaufmann, 1996.

[10] N. Weste and D. Harris. *CMOS VLSI Design A Circuits and Systems Perspective (3rd Edition).* Addison Wesley, 2004.