

The Pipelined Processor
ECE 312 Machine Problem 3

Tim Bergfeld and John Strologas

November 26, 1997

The Pipelined Processor

1 INTRODUCTION

Previously we designed a processor that executes a subset of the MIPS instruction set. This previous processor executed an instruction in an average of 4.1 cycles. While the number of cycles per instruction (CPI) is acceptable, we now look at implementing a pipelined processor in an attempt to increase the performance. This performance increase is due solely to the change in how the processor executes each instruction. For the nonpipelined processor, one instruction is fetched and processed to completion before the next instruction is obtained from memory. When doing sequential processing we allow some parts of the processor to remain idle. As we move to a pipelined case we have multiple instructions executing at the same time in different parts of the processor.

For example consider an assembly line. In sequential processing we start and complete one car before beginning the next. To increase the number of cars coming off of the assembly line, industry has implemented a version of a pipeline. One car has its frame assembled. Another has its motor installed while a third is equipped with tires. Still another has the interior assembled. In this way, four cars are being processed at any given instant in time. When the cars are done with one stage, they move on to the next. Hence, even though each car still takes the same amount of time to assemble, the number of cars completed by the assembly line is four times as many as the sequential case. In a similar way, we decrease the CPI for our processor by pipelining the execution of the instructions.

In order to explain the changes from the previous design, we start by describing the five stages of the pipelined processor and what is done in each stage. We also describe the instructions which are implemented and the register transfers associated with them. We then turn our attention to a calculation of the resulting performance of our processor and the impact some of the design decisions have upon the performance we can achieve. Finally, we turn our attention to the branches that cause problems

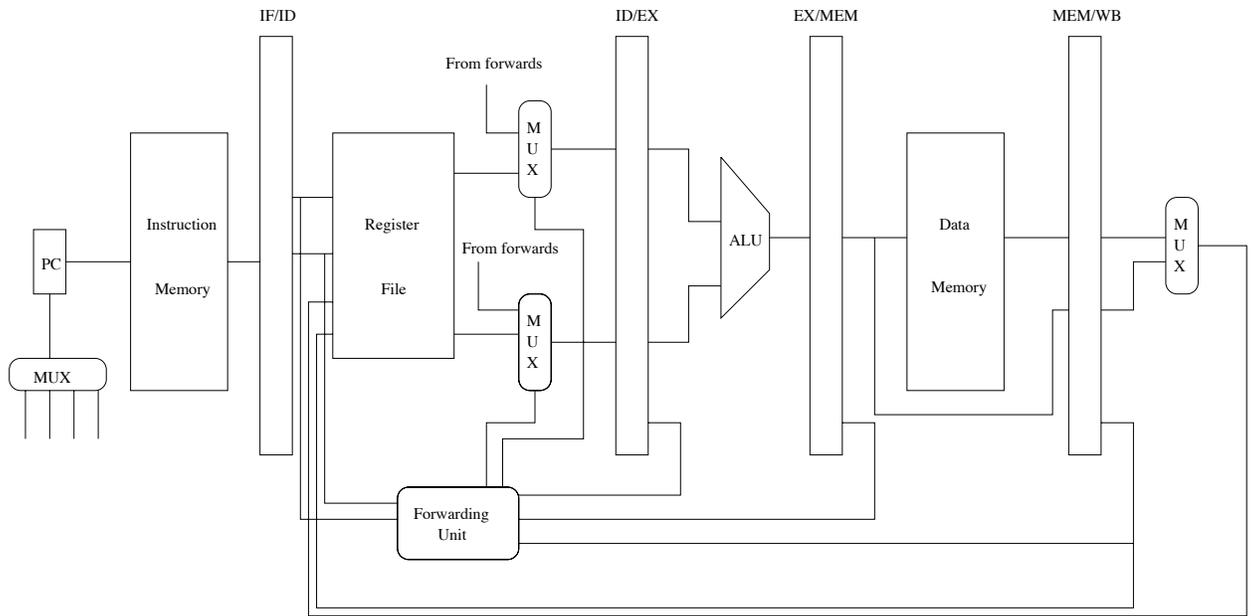


Figure 1: The five stages of the pipelined processor.

in the execution of the pipeline and explain how they are resolved and why we chose to resolve them with one delay slot.

2 DATAPATH

The five stages in our pipelined processor are the Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM) and Write Back (WB) stages, as shown in Figure 1. We latch the information which is passed between the stages to allow the clock cycle to be short. In the IF stage (Figure 2) we read the instruction at the address given by the program counter. The Instruction Memory is separated from the Data Memory allowing us to get the instruction-word while a previous instruction accesses the Data Memory. The Multiplexor before the program counter (PC) is used to select the address of the next instruction as either the current address plus four or the target instruction of a jump or a taken branch.

The ID stage is the most complex in our design as we see in Figure 3. Because we want to make our processor as fast as possible, we use the extra time available after

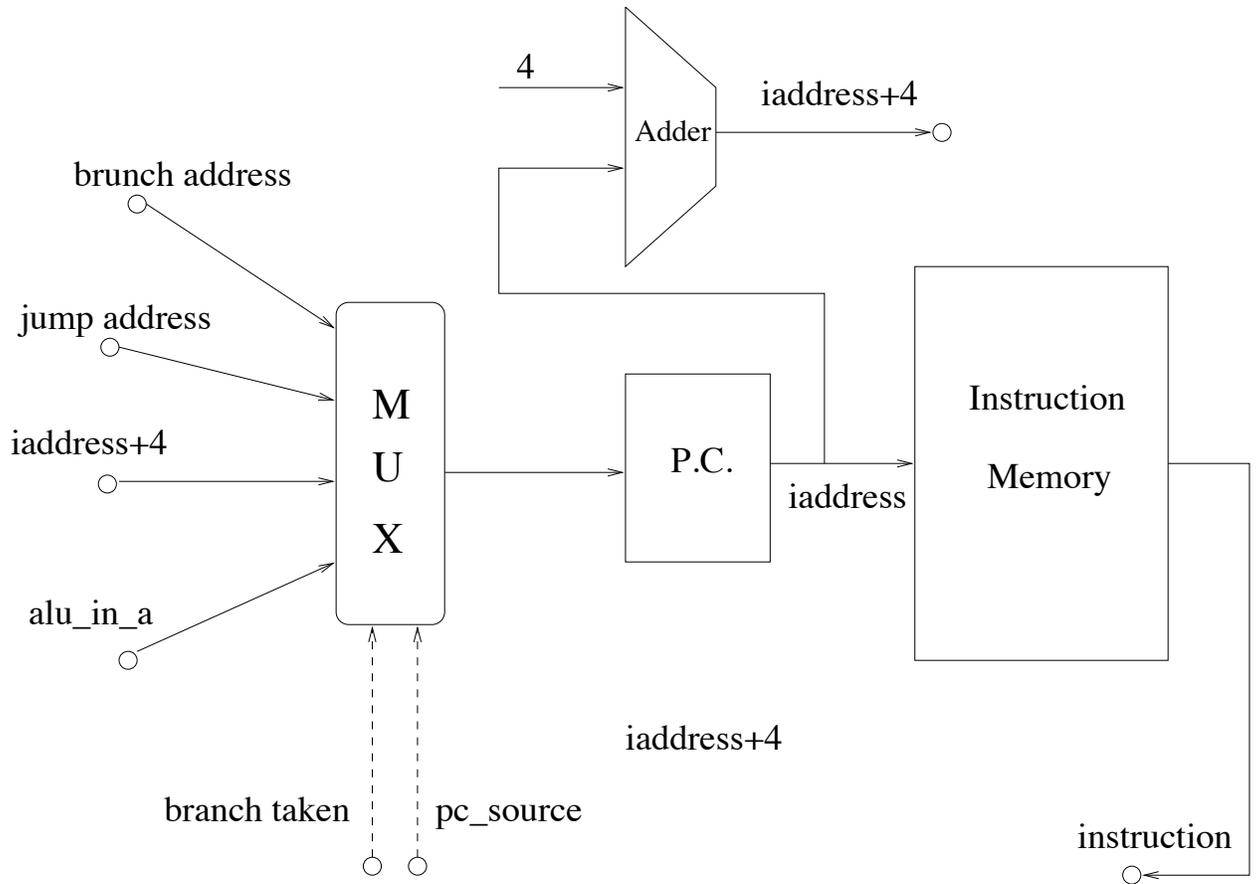


Figure 2: The Instruction Fetch stage.

reading from the register file to resolve the branches and process the forwarding. We take advantage of the extra time we have after reading the Register File and place the multiplexors which select the inputs to the arithmetic and logic unit (ALU) in the ID stage. This choice requires the forwarding unit, which forwards data from the previous instructions to avoid data hazards, be placed in this stage. Data hazards occur whenever an instruction uses the output of a previous instruction which has not yet been written back into the register file. The way we solve this problem is to forward the data from the latches following the stage where the data word is currently stored. The forwarding of data words is controlled by the forwarding unit. However, there are times when the data has not been calculated before it is needed. Thus, we must stall the pipeline in an interlock state and wait for the data to be calculated. The interlocks are discussed in more detail when we describe the performance of our pipeline. Finally the branch detection unit resides here, which compares the two registers, or the forwarded data, for equality and branches to a new instruction based upon the result. All of the control signals used by the pipeline are derived from the instruction-word by the Control Unit in this stage. The sign/zero extension unit allows for the immediate field to be extended for the ALU. The address of the current instruction can also be sent though the processor so it can be written into the register file.

The EX stage, shown in Figure 4, is simpler and faster with the multiplexors and the Forwarding Unit in the ID stage. The input data to the latch are the ALU inputs and the control. The outputs to the next stage are the ALU output, the control signals for the next stages and one of the inputs to handle the case of a memory store instruction. The ALU computes the result of the operation specified by the control signals generated by the control unit. We also have one multiplexor on each input in this stage. These multiplexors allows us to forward the ALU output from the previous instruction.

The MEM stage is described by Figure 5. In the MEM stage, the ALU output gives as the data memory address, while the data to be written are passed as a second 32-bit word. In case of a register load instruction, the data read from memory are

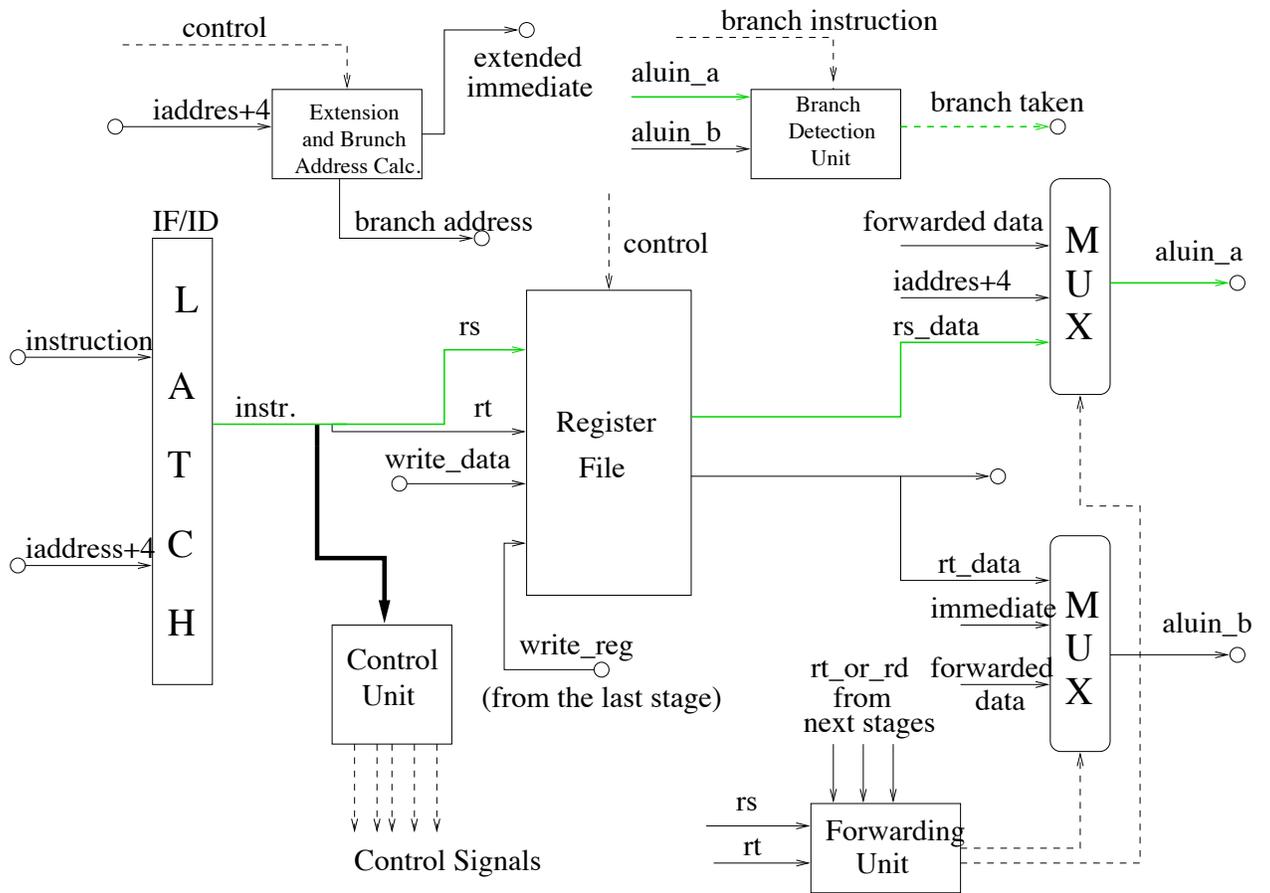


Figure 3: The Instruction Decode stage. The highlighted part of the diagram is the critical path for this stage.

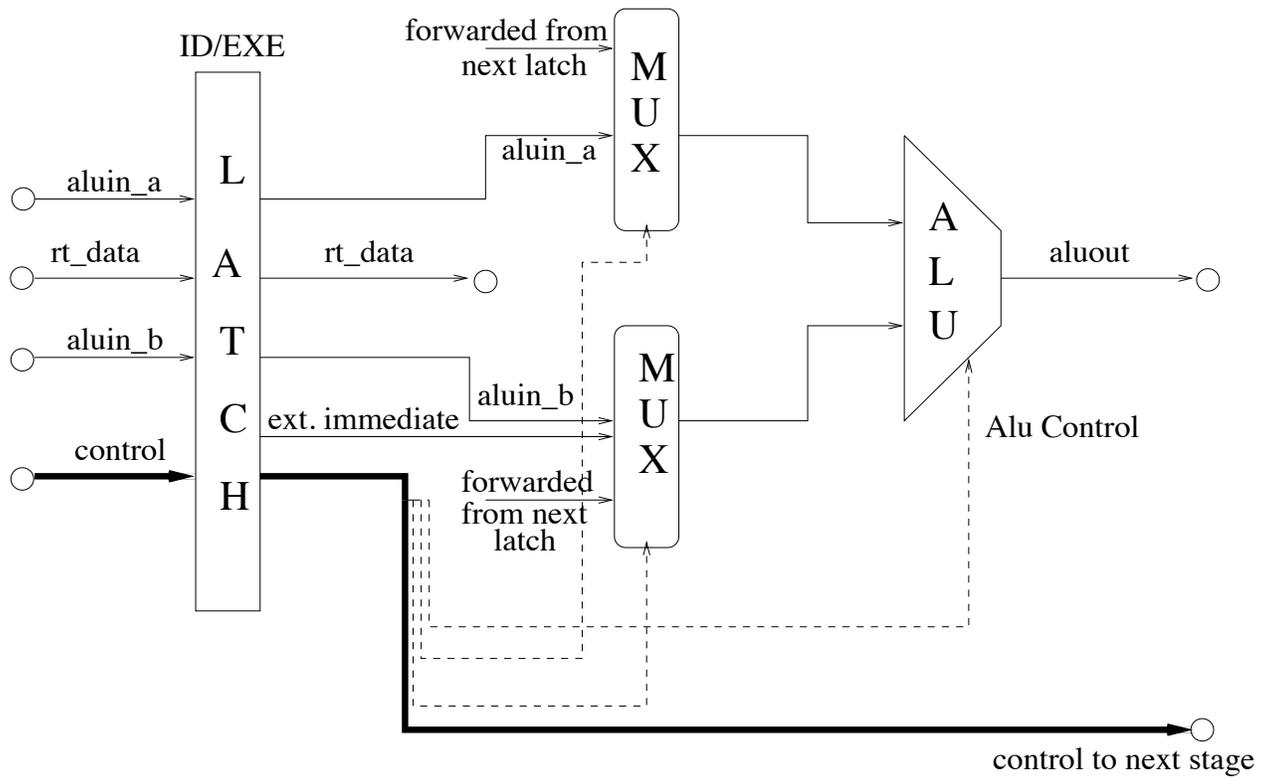


Figure 4: The Execution stage.

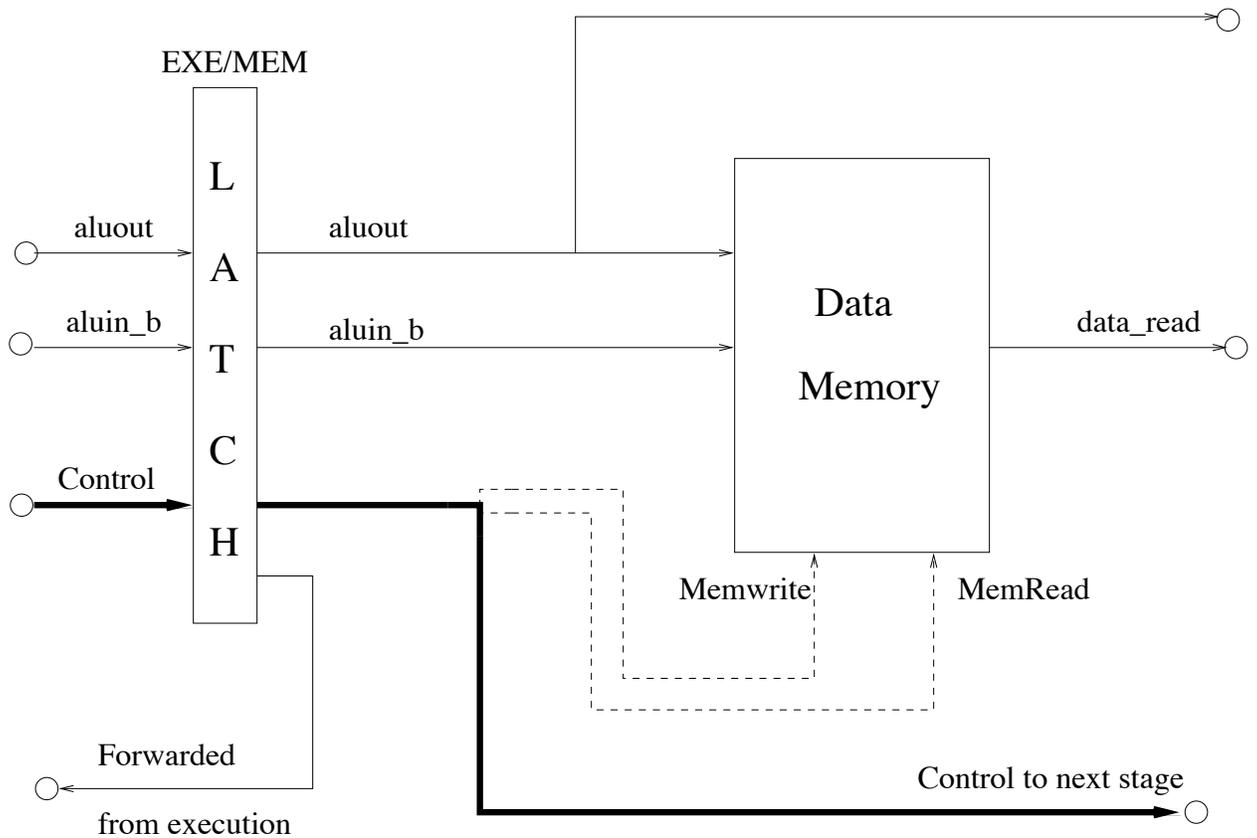


Figure 5: The Memory Access stage.

sent to the next stage to be written into the register file. We also allow the data word from the ALU output to be forwarded back to the Decode stage or the EX stage.

The final stage is the WB stage (Figure 6), in which we update the register file. It is the only stage in the pipeline where we write to the register file. We use the multiplexor to select data written as either the ALU output in case of R-type instructions or the data read from the memory in case of a register load instruction. The control signals gives us the register number to write, a write control line to the register file and the multiplexor control signal.

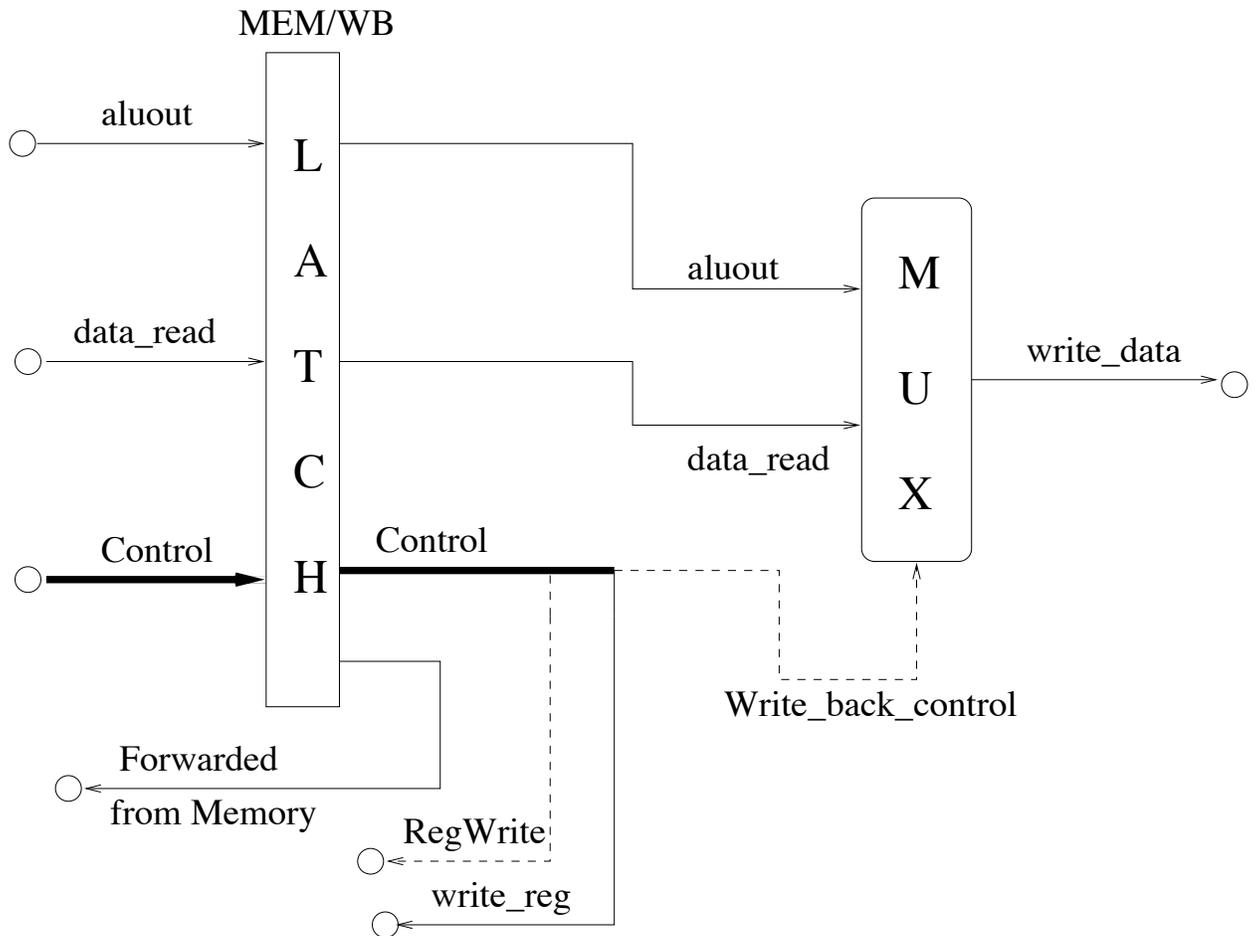


Figure 6: The Write Back stage.

3 THE INSTRUCTION SET

Our processor can execute 19 instructions. Before describing their details, we define some useful fields in the instruction word. The *rs*, *rt* and *rd* fields are the address of the register given by bits (25 to 21), (20 to 16) and (15 to 11), respectively. The immediate field is bit 15 to bit 0 of the instruction word. The target field is bits 25 through 0. The description of the register transfers for each instruction can be found in the Appendix. Here we describe the instructions which have been implemented.

3.1 R-type instructions

The R-type instructions have two input arguments and produce an output which is a function of the inputs. Our instruction set includes the addition, subtraction, the logical OR, AND, shift right and shift left and the instruction which sets the output register to 1 if one of the input arguments is less than the other. These instructions are named *add*, *sub*, *or*, *and*, *srl*, *sll* and *slt* respectively.

In order to implement an R-instruction, we fetch it from memory. We then decode the instruction and produce the proper control signals. The ALU inputs are in registers *rs* and *rt*, which are read while we produce the ALU control signals. In the EX stage, we execute the instruction by sending the data to the ALU and selecting the proper operation using the control signals produced in the previous stage. We do not have any memory access for these instructions, so we bypass the MEM stage. Finally, we write the result from the ALU to register *rd* during the WB stage.

3.2 I-type instructions

The process is the same with the R-type instructions, with the only difference being one of the inputs to the ALU is the sign-extended or the zero-extended immediate field. Additionally, we set the destination register address to *rt* instead of *rd*. For the *addi*, *slti* instructions we use a sign extension while we zero-extend the immediate field for the *andi* and *ori* instructions. The only instruction which does not have an R-type equivalent is the *lui*, load upper immediate. It loads the immediate field into the upper halfword of register *rt*.

3.3 Memory Access Instructions

The memory access instructions are the store word, `sw`, and the load word, `lw`, which write into and read from the memory respectively. After the processor fetches the instruction, the ID stage sends the `rs` register contents, the sign-extended immediate and the contents of register `rt` to the next stage. At the EX stage, we add the contents of the `rs` register and the sign extended immediate to obtain the memory address. The MEM stage is the most important for this set of instructions. For the load (store) instruction, we use the address calculated in the previous stage to read (write) data from (to) the memory. Finally, in the case of a load command, we write the output of the memory to the register `rt` in the WB stage.

3.4 Jump Instructions

The set of jump instructions contains types of jumps: unconditional jump to the instruction at the address given in the target field, `j`; an unconditional jump to the instruction at the address contained in the register `rs`, `jr`; and a jump and link instruction which jumps to the target address and stores the address of the next instruction in register `R31`, `jal`. The first two types of jumps only use the first two stages of our pipeline, since we just need to fetch the instruction and calculate the next address which is sent to the program counter by the decode stage. For the `jal` instruction, we use the Execute stage to calculate the next address and the WB stage to update register `R31`.

3.5 Branch Instructions

The two branch instructions, `beq` and `bne`, branch to the address specified by the offset field if the contents of registers `rs` and `rt` are equal or not equal respectively. Again we just need the first two stages of the processor to fetch the instruction, calculate the branch address and decide if the branch is taken or not. Due to the pipelining when we decide if a branch or a jump is taken, the next instruction has already been fetched from memory. This next instruction is the delay slot which may

Instruction Group	Frequency	Cycles	Interlock Contribution	Delay Slot Contribution	Nonpipelined CPI
ALU Operations	30%	1.0	0.0	0.0	4.0
Immediate ALU	15%	1.0	0.0	0.0	4.0
beq, bne	12%	1.502	0.052	0.45	4.0
j	4%	1.45	0.0	0.45	3.0
jal	2%	1.45	0.0	0.45	3.0
jr	2%	1.502	0.052	0.45	3.0
lui	5%	1.0	0.0	0.0	4.0
lw	20%	1.05	0.05	0.0	5.0
sw	10%	1.0	0.0	0.0	5.0
Total	100%	1.107			4.1

Table 1: The performance measurement for the processor in terms of cycles per instruction and the contributions from interlocks and delay slots. The values for the nonpipelined processor are given for comparison and are based on the results of the previous machine problem.

or may not be filled with a useful instruction.

4 PERFORMANCE ANALYSIS

We can now compare the performance of the pipelined processor and the non-pipelined processor from the previous machine problem to demonstrate the gains resulting from pipelining. While exact performance can only be measured by the time required to execute specific programs, we can approximate the time with the following equation:

$$\text{Time} = \text{CPI} \times \text{Clock Period} \times \text{Number of Instructions},$$

where the CPI refers to the weighted average number of cycles necessary for an instruction to complete execution. We compute the weighted average by weighting each

instruction duration by the frequency of that instruction in the program. However when comparing the processor without a specific program we must make assumptions about the frequency of each type of instruction. The frequency assumptions are shown in Table 1. The number of cycles for each instruction is easily calculated for a nonpipelined case, because only one instruction is executing at a time. The resulting CPI for the processor designed in the previous machine problem is shown in Table 1 for comparison.

Next, we need to calculate the CPI for the pipelined case. If the program was limited to executing ALU operations the number of cycles per instruction would be one in all cases. However branches, load and stores can cause problems with the execution of the program. With the choices already made (i.e., five stages in the pipeline and branch resolution in the second stage with one delay slot), we must take into account the effects of the delay slot and possible interlocks on a pipeline which will occur. To account for these complications, we increase the number of cycles to execute a branch instruction by the average number of delay slots which will go unfilled. This increase is shown in Table 1 as the Delay Slot contribution. We assume that 55% of the delay slots can be filled by useful instructions which will not delay the processor; hence, we only increase the CPI for branches and jumps by 45%.

The last contribution to the CPI is the resulting delay incurred when an interlock occurs. Interlocks occur when the data needed for an instruction before it has been calculated. This problem in the pipeline will occur when a branch instruction is dependent upon the previous instruction's ALU operation. Since the branch instruction is at the second stage of the pipeline at the same time that the previous instruction is at the execution stage, we cannot resolve the branch until the next cycle when the data can be forwarded back to the ID stage. The interlocks will also be invoked if the branch or jump register instruction needs the result from a memory load instruction which is two instructions before the branch. In these cases we stall the pipeline by rereading the next instruction and not changing the IF/ID latches which pass the instruction from the IF stage to the ID stage. The other stages will execute as normal with a useless instruction passed into the next stage. Thus, we

must stall the pipeline in an interlock state and wait for the data to be calculated. The contribution from interlocks in Table 1 is due to one instruction having to wait for the execution of the previous instruction before it can proceed. Fortunately, the processor doesn't have to wait very often. We assume that only 5 % of the time do we have an instruction which is dependent upon the previous instruction and only one percent of the time is it dependent upon the execution of the instruction before the previous instruction. These assumptions allow us to calculate the contributions to the CPI for the interlocks.

For example, as shown in Figure 1, we will have the results of a load word instruction at the end of stage four. If the next instruction is dependent upon that word from memory we will incur a one cycle delay. To account for this delay, the CPI of a load instruction is increased by the frequency of this combination, 5 %. To force correct execution of these combinations without extra hardware, we require the compiler to insert an appropriate noop instruction for this case. The other chance for an interlock occurs when an instruction dependent upon the previous instruction requires evaluation in the second stage, i.e., a branch or a jump. These interlocks, which are enforced by the hardware itself, cause an increase in the CPI for those instructions by 5 %. Unfortunately, there is one final case that can occur. If an instruction that is dependent upon the results of a load occurring two instructions previous requires evaluation in the second stage, then we will have to stall the pipeline. This last type of interlock is accounted for by adding a contribution to the CPI of 0.002 cycles. The additional contribution was calculated by assuming a 1 % probability of an instruction having a dependent instruction two instructions ahead and a 20 % chance of a given instruction being a load word.

After having the average number of cycles per instruction, we must examine the clock period and its effect upon the performance. Ideally, we would have a very short clock cycle; however, the hardware takes time to evaluate each stage. To choose the clock period, we look at each stage and evaluate how long each stage will take to finish executing. We must choose the clock period to be at least the time required for the slowest stage of the pipeline. This maximum occurs in both stage one (the

Stage	Critical Path	Time
Instruction Fetch	Register, Memory	14 ns
Decode	Register, Register File, Multiplexor, Adder, Multiplexor	14 ns
Execution	Register, Multiplexor, ALU	12 ns
Data Memory	Register, Memory	14 ns
Write Back	Register, Multiplexor, Register file	11 ns

Table 2: The critical path delay for each of the processor stages. The critical path in the decode stage occurs in the case of a branch statement. All other critical paths are for each type of instruction.

instruction fetch) and stage four (the data memory). The clock period is the sum of the delay of the register which contains the address for the appropriate memory and the memory response time. With the requirements given by the hardware for this processor, we have a 14 ns clock period since the memory requires 12 ns and the registers require an additional 2 ns. Unfortunately, we cannot reduce this clock period if we insist upon fetching one instruction every clock cycle. As we can see in Table 1, we have achieved a performance increase of 3.7 assuming identical clock cycles for the two processors.

5 OTHER DESIGN ALTERNATIVES

There are other options for the design of the processor. Because the data fetch and ALU execution stages do not always require the full 14 ns to complete execution, we can consider combining them into one stage. This combination would allow us to use the ALU to test the equality for the branch execution. If we were to implement this change, the interlocks occurring with the branching and jump register instruction which are currently evaluated in stage two would disappear. The CPI for those

instructions would then be 1.45 resulting in an average CPI of 1.1. Unfortunately, while the CPI would decrease, the time required for execution of the combined stage would increase to at least 16 ns. The time would be the sum of 2 ns for the instruction register, 4 ns to read the register file, 9 ns for the ALU to obtain a result and an extra 1 ns for a multiplexor which would have to select either the immediate or register file data to go into the ALU. The additional time would give 17.6 ns per instruction up from 15.5 ns per instruction. We see why a four-stage design would degrade the performance of the processor.

We can also consider other alternatives to the dual memory structure. In the pipelined processor, we separate the instruction and data memories to allow for the possibility of reading the next instruction and reading or writing to the data memory. If we were to only use one memory for both data and instructions as we did in the previous processor, then a read or write to memory would stall the pipeline for one cycle during the memory access. The cost of the stall in the pipeline would be to increase the average CPI to 1.4. Because this increase is unacceptably large, we have chosen to implement the dual memory structure.

After deciding to use a five-stage pipeline for our processor design and before implementing any part of our design, we had to choose the stage in which the branches will be resolved and the way we will handle them either using squashing or delay slots.

If we have the branch detection unit at the execution stage, giving two delay slots, the CPI for a branch will be 2.0 for the squashing case, in which two stall cycles will occur when the branch is not taken. We assume 50% of the branches are taken. Similarly if we use two delay slots, where the first is filled 75% of the time and the second only 20%, the CPI becomes 2.05.

If the branch detection unit is in the decoding stage with only one delay slot, the CPI for a branch will be 1.55 for the squashing case where there is one bubble 50% of the time and 5% of the time we have an interlock stall. Finally, for the Delay Slots design with the branch detection unit in the decoding stage, the CPI for a branch is 1.502 as described above.

Branch Unit Position	Squashing	Delay Slots
Execution stage	2.0	2.05
Decode stage	1.552	1.502

Table 3: The number of cycles per instruction for branch instructions. The numbers presented in this table have accounted for interlocks and the control hazards.

We compare these numbers by looking at Table 3. We observe that the design giving the lowest CPI for a branch instruction is the one-delay-slot design, which we have chosen. One should also point out that the jump instructions will have a similar CPI.

6 CONCLUSION

We have presented our design of a five-stages pipelined processor with one delay slot. The branch detection and the forwarding unit as well as the multiplexors which control the inputs to the ALU reside in the second stage. This allows us to minimize the critical path and make our processor as fast as possible, with an expected clock cycle close to 14 ns. The data hazards are controlled by the forwarding unit and interlocks which may be inserted. To avoid control hazards we use a single delay slot. We also outlined all the stages of the pipeline and provided the Register Transfer Language descriptions of the 19 instructions of our set. We have given a justification for the design we have chosen and described the factor of 3.7 increase in performance obtained from a pipelined design.

7 APPENDIX

For completeness, we present the Register Transfer Language descriptions of the stages of the pipeline in tabular form. In each of the following tables, the implemented instructions are broken down into the register transfers which will take place at each stage.

R-type instructions (add, sub, or, and, slt, srl, sll)		
Pipeline Stage	RT instructions	Control Signals
IF	$IR \leftarrow M(PC)$	pc_src
ID	$ROUT1 \leftarrow rs, ROUT2 \leftarrow rt$	aluin_a_mux=rs, aluin_b_mux=rt rd_rt_mux=rd
EX	$ALUOUT \leftarrow ROUT1 (op) ROUT2$	ALUop
MEM	-	-
WB	$rd \leftarrow ALUOUT$	wb_mux.control=ALUOUT, rd_rt_mux=rd, regwrite

Table 1: R-type instructions (add, sub, or, and, slt, srl, sll).

I-type instructions (addi, ori, andi, slti, lui)		
Pipeline Stage	RT instructions	Control Signals
IF	$IR \leftarrow M(PC)$	pc_src
ID	$ROUT1 \leftarrow rs,$ $ROUT2 \leftarrow \text{extended immediate}$ (zero extended for ori, andi)	aluin_a_mux=rs, aluin_b_mux=extended rd_rt_mux=rt
EX	$ALUOUT \leftarrow ROUT1 (op) ROUT2$	ALUop
MEM	-	-
WB	$rt \leftarrow ALUOUT$	wb_mux_control=ALUOUT, rd_rt_mux=rt, regwrite

Table 2: I-type instructions (addi, ori, andi, slti, lui).

lw, sw		
Pipeline Stage	RT instructions	Control Signals
IF	$IR \leftarrow M(PC)$	pc_src
ID	$ROUT1 \leftarrow rs,$ $ROUT2 \leftarrow \text{sign extended immediate}$	aluin_a_mux=rs, aluin_b_mux=sign extended rd_rt_mux=rt
EX	$ALUOUT \leftarrow ROUT1 + ROUT2$	ALUop=+
MEM	Memory(ALUOUT) \leftarrow ALUIN_B (sw) Memory(ALUOUT) \Rightarrow WB latch (lw)	MemWrite (sw) MemRead (lw)
WB	$rt \leftarrow \text{WB latch}$	wb_mux_control=DATA , rd_rt_mux=rt, regwrite (all for lw)

Table 3: Memory access instructions (lw, sw).

j		
Pipeline Stage	RT instructions	Control Signals
IF	$IR \leftarrow M(PC)$	pc_src
ID	$PC \leftarrow \text{target}$	pc_src=jump
EX	-	-
MEM	-	-
WB	-	-

Table 4: Jump instruction.

jr		
Pipeline Stage	RT instructions	Control Signals
IF	$IR \leftarrow M(PC)$	pc_src
ID	$ROUT1 \leftarrow rs$ $PC \leftarrow ROUT1$	pc_src=jr
EX	-	-
MEM	-	-
WB	-	-

Table 5: The jump register instruction.

jal		
Pipeline Stage	RT instructions	Control Signals
IF	$IR \leftarrow M(PC)$	pc_src
ID	$PC \leftarrow target$	pc_src=jal
EX	$ALUOUT \leftarrow (PC+4)+4$	ALUop=+
MEM	-	-
WB	$R31 \leftarrow ALUOUT$	regwrite, reg=31

Table 6: The jump and link instruction.

beq, bne		
Pipeline Stage	RT instructions	Control Signals
IF	$IR \leftarrow M(PC)$	pc_src
ID	$PC \leftarrow sign_ext \ll 2 + PC + 4$	branch_detection=branch_taken(or not)
EX	-	-
MEM	-	-
WB	-	-

Table 7: Branch instructions (beq, bne).