Design and Implementation of a Superscalar Microprocessor

Alex Olson
aolson1@mail.utexas.edu
Tom Hartin
hartin@ece.utexas.edu

Vindhya Rajan
vindhya_rajan@yahoo.com
Sean Leather
leather@ece.utexas.edu

ABSTRACT

Superscalar processors are machines that are designed to fetch and issue multiple instructions every clock cycle. This paper discusses the complexity in the transition from non-pipelined processors to pipelined processors and the transition from pipelined processors to superscalar processors. We begin with the discussion of an implementation of a non-pipelined processor. Secondly, we discuss the process of converting it into a pipelined processor. Finally, we provide the design details of all the phases of a superscalar processor, including the performance improvement achieved by these transitions.

Keywords

superscalar, pipeline, design flow, LC-3b

1. INTRODUCTION

Over the last 50 years, microprocessors have increased tremendously in performance. The metric of cycles per instruction (CPI) has often been used to compare the various designs. CPI has fallen with every new design due to improved technology as well as microarchitecture techniques such as the pipeline. Building upon this, engineers have increased the number of instructions that can advance through a pipeline at a time. These superscalar processors employ multiple functional units to further increase throughput. As a result of this trend in design, the complexity of these processors has increased as much or more than the performance has improved. This paper attempts to show the improvement in performance of a superscalar processor over a simple design and a scalar, pipeline design. We also demonstrate the complexity of realizing a superscalar processor by providing an example of the design flow.

1.1 Background

Pipelining is an implementation technique where multiple instructions are overlapped such that they are executed simultaneously in the processor. Hennessy and Patterson[1] compare a pipeline to an assembly line with many steps: each step in a pipeline is called a pipe stage. All the stages in a pipeline advance at the same time and this reduces the execution time of the instructions, which can be considered as a reduction in the overall CPI. Since the instructions are evaluated in parallel, this overlap between instructions is called Instruction level parallelism - ILP.

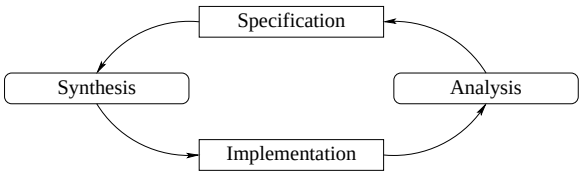
In the early 80s, this technique was used primarily in supercomputers and large mainframes. By the early 90s almost all processors embraced pipelining. The first general purpose pipelined processor was IBM 7030. It was a four-stage pipeline designed with a goal of increasing the performance of its predecessor, the IBM 704 by a factor of hundred to two hundred[7]. The RISC processors were originally designed with pipelining in mind[1]. Since then many papers have been published discussing the benefits of pipelining, examining instruction issue, exception handling, pipeline depth, etc.

The techniques that were implemented in the processors had a common goal of exploiting the parallelism of an application. Since the parallelism available within a block is small, ILP is exploited across multiple blocks[1]. These techniques are either done statically by the compiler or dynamically by the hardware. Alternative methods, like vector processing were developed to increase the throughput of the processor. Although vector processing preceded processors that exploit ILP, it would not be an understatement to say that the latter has replaced Vector processors. Hennessey and Patterson[1] speculate that vector processors will gain popularity again but, not as General-purpose processors.

1.2 Motivation

In spite of all the techniques involved in implementing a pipeline, scalar processors, by definition, are limited to a throughput of only one instruction per cycle. Another major limitation of scalar pipelines is that they are typically rigid. Different instructions have different latencies and different hardware requirements, but the scalar pipeline forces unification of all the instruction types into a single pipeline. Since the instructions advance in a lock step fashion the throughput of the processor suffers. [6] For greater perfor-

mance, these limitations had to be overcome, and a solution to this problem is the Superscalar pipeline. Multiple instructions can be fetched and executed in parallel in this pipeline.



The process of designing and implementing a supercalar machine is both complex and time-intensive. Though our aspiration is to design a superscalar processor starting with simple non pipelined processor, it was also our goal that we experience the design flow depicted above. This engineering design is an iterative process of Specification, Analysis, Implementation and Synthesis. We decided to use the non-pipelined LC-3b processor[3] architected by Patt et al., as the baseline processor and to build the pipelined version and the superscalar pipelined processor based on it. In context of the engineering design, the specification would be the Instruction Set Architecture(ISA) of LC-3b, which made the specification a fixed entity. So, our iterative process technically involved only Analysis, Implementation and Synthesis.

1.3 Overview

The following sections discuss the design details of each implementation of the LC-3b. Section 2 describes the LC-3b ISA and its basic design. In Section 3, we define the design process for the pipeline architecture as well as the resulting design. We then build upon that in Section 4 with the description of our superscalar, pipelined design. Section 6 contains physical and performance comparisons of the designs. After that, we provide our concluding remarks in Section 7. The products of our work include: a formal design for scalar and superscalar pipelined processors, synthesiable Verilog realizations of each, and performance comparisons.

2. BASELINE DESIGN

2.1 Methodology

The LC-3b is a very simple RISC-like architecture used for educational purposes. Its 16-bit, fixed-length instruction word supports 22 different basic instructions, which are described in Table 1. No complex instructions such as multiply or divide are included in this ISA. Logic and arithmetic instructions use only direct and register addressing; memory and control instructions use register indirect and direct (PC-relative) modes.

In this ISA, memory is byte-addressable with a 16-bit address, and words are stored as little endian. Loads and stores are the only instructions that access the memory.

All arithmetic, logical, and memory instructions that write to one of the eight 16-bit, general-purpose registers also writes to the CC register. It holds three bits: Positive (P), Zero (Z), and Negative (N) that are set depending on the value stored. The conditional branch instruction (br) use the CC register to determine whether to jump to the target or not.

2.2 Implementation

All realizations of the LC-3b will perform the basic instructions as described by Patt[4] except for the TRAP instruction. Our implementations will not support interrupts or exceptions; however, we do support the later modification of precise exception without radical redesign.

In order to see how the performance improves with later versions of the processor, we have a baseline implementation. This was derived from Patt's description of a basic state machine and microarchitecture[3]. It can be split into two basic defining blocks of control and the data path.

Control is done through microcode. A ROM-like structure called the control store contains the microinstruction for every valid state of the machine. Each state uses the collection of control signals in the microinstruction to address the next microinstruction in the control store as well as to enable, disable, or select some functionality in the data path.

3. PIPELINE DESIGN

3.1 Methodology

Although the baseline version requires very simple (and minimal) hardware, it does not give very good performance. It requires several cycles to execute one instruction, and many of its functional units will be idle for a significant amount of time. Although our final goal is to create a superscalar pipelined processor, we see creating a scalar pipelined processor as good starting point. The general issues that the hardware needs to handle in the pipeline stages are:

Instruction Fetching An n-way, pipelined processor should be able to fetch n instructions on every clock cycle. Methods to handle misalignment of the instructions being fetched and methods to predict branches are implemented to maximize intruction bandwidth.

Instruction Decoding This step identifies the dependencies between instructions, as well as the requirements of each instruction (register access, memory, etc).

Issue This step decides what instructions should be issued and when to issue them.

Execution Functional units carry out the tasks to be performed by each instruction.

Completion and Retiring Responsible for committing the instructions in program order. In-order committal is necessary for architecturally correct execution and for also precise exceptions[6].

According to Shen and Lipasti, there are 3 objectives of designing a pipelined architecture[6]. These "pipelining idealisms" are:

- Uniform Subcomputations
- Identical computations
- Independent computations

In the following sections, we describe how we apply this process to our design.

Table 1: Categorization of LC-3b instructions.

Instruction Type	Examples	Description
Logic / Arith.	ADD, AND, SHF, XOR	Logic or arithmetic operations
Data movement	LDB, STB, LDW, STW	Transfer a byte or word between a register and memory
Control	BR, JMP, JSR, TRAP	(Un)conditional branches, subroutine jumps, traps

3.1.1 Uniform Subcomputations

In the ideal case, all computations can be evenly partitoned into k (for $k > 1$) uniform latency sub computations. In reality, this is not always possible. The minimum clock period of a pipeline design is determined by the stage with the longest latency. One of the primary challenges in pipeline design is to minimize *internal fragmentation* or the differences in latency between the stages through effective stage quantization.

Tables 2 and 3 show the basic tasks that need to be performed with each type of instruction and their associated latencies. In order to determine latencies of these subcomputations, we wrote a few modules in Verilog and synthesized them using 0.18 micron technology. As shown, the ALU, shifter, and register file all have fairly simliar latencies, which indicates that pipelining can be highly beneficial for this ISA. We do not synthesize memory and we assume zero-latency accesses; however, we did test various memory latencies delays in our implementations.

3.1.2 Identical Computations

In a multi-function pipeline, it is nearly impossible to have identical computations. Often some stages are unused by an instruction (e.g. a memory access stage for an add operation), and this creates an inefficiency called *external fragmentation*. To minimize external fragmentation, different resource requirements must be combined according to different instruction types.

All of the LC-3b instructions share the same computations of fetch and decode. Some share other computations such as register file access (e.g. AND, XOR, and JSRR all source the register file) and memory access (e.g. LDW and LDB both read and STW and STB both write to memory). These are shown in Table 2. Due to this quality, we can combine some operations into the same stage. For example, branch resolution occurs in the same stage as memory accesses. Also, instruction decoding and register reading can done in the same stage.

3.1.3 Independent Computations

The third idealism is that the computations at any stage in the pipeline are independent of each other. In other words, a later computation should not wait for the result of an earlier one. In our case, we must support control and data dependencies; therefore, all mechanisms to resolve these should be implemented in the most efficient way possible.

For maximum performance, we include all possible forwarding paths for data. We also implement branch prediction to reduce the effects of control dependencies. More on this will be discussed in Section 3.2.

3.1.4 Unification

We grouped the subcomputations in Table 2 into five stages, based on the similarity in the list of subcomputations for each instruction.

IF: Instruction Fetch Instructions are read from the memory in this stage. Since the address space of this ISA is only 16 bits, and most programs for this ISA are very small, we will assume that fetched instructions are always available by the end of the clock cycle. We also do branch prediction in this stage.

ID: Instruction Decode / Register File Read In this stage, we decode an instruction, check for data dependencies, and read the source operands from the register file. The instruction formats are very regular; so, it takes minimal logic to derive the register addresses. This allows us to do the decode and read operations in parallel.

EX: Execute In this stage, we perform any arithmetic, logical, shift, operations. For memory and control instructions, address generation is also done.

ME: Memory Access In this stage, we perform memory read/write operations. We are only concerned with the latency of the supporting logic and not memory latency itself. For byte operations, sign extensions are done in this stage. Conditional branches read the (CC) condition-code register, and all control instructions are resolved. When mispredictions occur, the PC is loaded with the correct target address on the next cycle.

WB: Write Back In this stage, the register file and CC register are updated for those instructions that require it.

3.2 Implementation

In the following sections, we discuss the implementation of our scalar pipeline design. For reference, Figure 1 shows how the stages are arranged as well as the general flow. At present, we know of only one other pipelined implementation of this ISA. An outline of it is specified in an unpublished document by Patt [5]. It is interesting to note that the process of following Shen's methodolgy for pipeline design led us to a simliar structure as found by Patt. However, our design and implementation make several novel contributions, including branch prediction and data forwarding. Patt's pipeline specification specifies that the pipeline stalls until the branch is resolved, or until a data-dependency no longer exists. Toward the end of this paper, we show the performance comparison between all LC-3b implementations, including how forwarding and branch prediction improve performance.

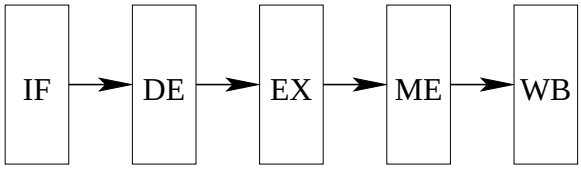


Figure 1: Our design, a 5 stage pipeline

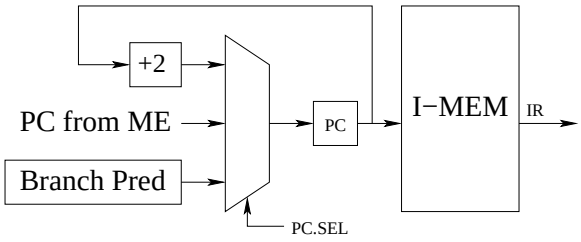


Figure 2: Instruction Fetch (IF) stage of the scalar pipeline design.

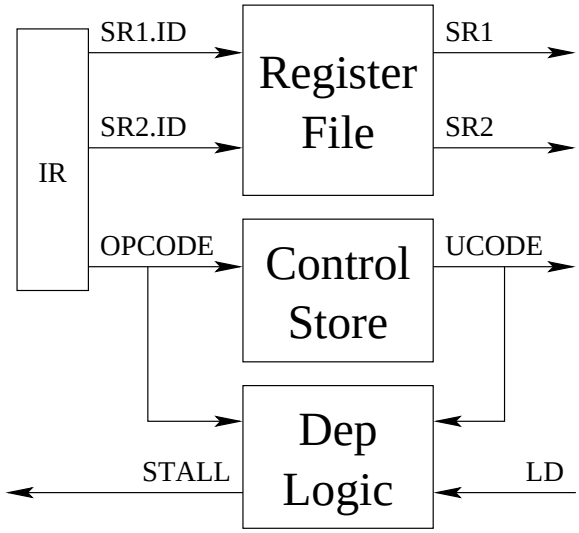


Figure 3: Decode (DE) stage of the scalar pipeline design.

3.2.1 IF: Fetch

This stage is fairly simple as is shown in figure 2. It contains the PC and is responsible for retrieving instruction from memory. Due to the small address space of this ISA (only 16 bits), and the size of our programs, we assume for simulation purposes that the memory is already ready on every cycle. Today, modern processors have primary caches of 64KBytes, so the idea of the entire system memory residing on the processor is not too far-fetched. On every clock cycle, the PC is updated with the value from the multiplexer shown. The PC.SEL signal is composed of one bit from the branch logic in the ME stage and one bit from the output of the branch predictor. The branch predictor is described in detail in 3.2.4. When a branch is mispredicted, the PC is loaded with the correct address, which will either be the next PC (NPC), or the branch target. If no branch is mispredicted, then the PC will be loaded with the predicted branch target of the recently fetched instruction. If neither of these are true, then the PC is simply incremented by one instruction word (2 bytes).

3.2.2 ID: Decode

In this stage, instruction decoding and register file access take place. Figure 3 shows a basic diagram of this stage. The control store, indexed by 6 bits of the instruction opcode, is a table that generates most of the control signals needed by the other stages. We implemented this as a 'casex' in Verilog. In reality, this would be synthesized as a hard-wired design. The control store's outputs include signals such as register-write enables, and ALU controls. The dependency logic, shown in the figure, has two functions. It stalls the IF stage when a dependent instruction is issued following a LDB/LDW instruction. This effectively inserts one bubble between the dependent instruction and the load instruction. The dependency logic also generates all forwarding signals for functional units in the EX stage.

3.2.3 EX: Execute

The Execute stage, shown in figure 4 is used by every instruction and is an important stage in our processor. Arithmetic and logic instructions do their main computations here. Also for load/store instructions, memory addresses are also calculated here. Lastly, control instructions also do their address calculations here.

As mentioned before, we implement all possible forwarding paths, in order to reduce the effects of data dependencies among instructions. We provide forwarding paths from ME to EX, and from WB to EX. It is important to note that all references of a stage name refer to the combinational logic within a stage. Hence forwarding 'from ME 'means to take the result from the output of the sequential buffer that follows the combinational logic of the EX stage. This terminology is slightly different than that used by Shen, but we feel it more accurately conveys the idea. Our forwarding paths allow a series of dependent instructions to be executed without stalling, provided no memory accesses are involved. If the instruction following a LDB/LDW instruction needs the result of the load, there is a 1 cycle stall, followed by a utilization of the WB-EX forwarding path. Simultaneous writes and reads to the register file and condition code registers are both possible and will produce valid results.

The 'SR1' and 'SR2'signals depicted in figure 4 represent the values of the operands. They may be the values obtained from the register file (on the previous cycle), or values provided from the forwarding logic just described.

3.2.4 ME: Memory Access

In this ISA, the ME stage requires the address calculated in the EX stage (for loads/stores), so no forwarding is possible nor beneficial. For branch resolution, the condition code register is read in this stage. Internally, the condition codel logic takes care of the case where the condition code register is being written and read simultaneously. Thus, no forwarding logic is needed.

Branch resolution in this stage consists of ANDing three bits from the instruction opcode with the three bits of the condi-

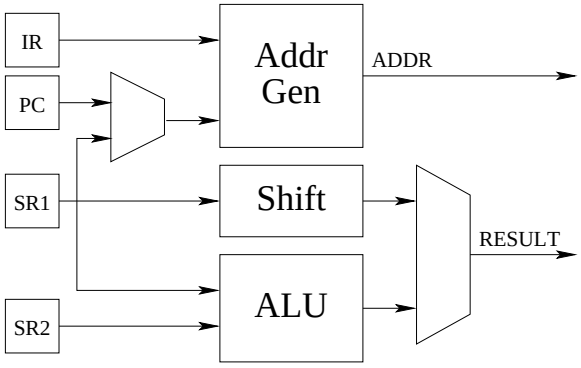


Figure 4: Execute (EX) stage of the scalar pipeline design.

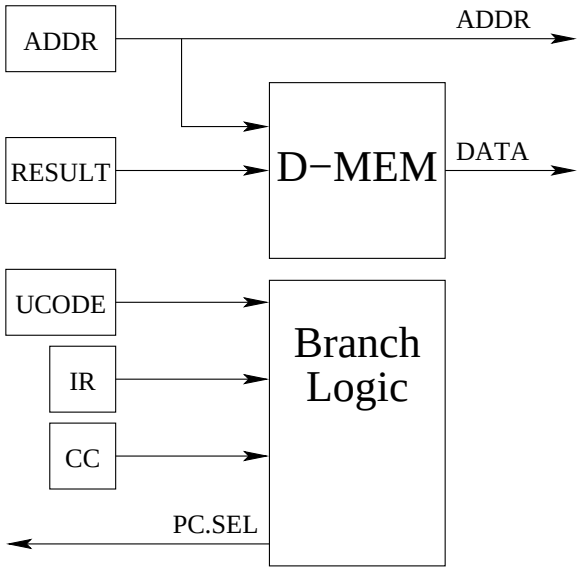


Figure 5: Memory (ME) stage of the scalar pipeline design.

tion code register, and seeing if the result is non-zero, which indicates a taken branch. The value of the prediction from the IF stage is carried through the pipeline with the rest of the signals associated with each instruction. If the ME stage sees that the predicted branch outcome and the actual branch outcome differ, then it asserts its PC.SEL line. This causes the instructions currently in IF, ID, and EX to be marked invalid (using combinational logic). On the next cycle, the correct PC will be fetched. Is a branch was mispredicted, and was actually taken, the PC will be will be updated with the branch target address that was computed during the EX stage. If the branch is mispredicted, and found to be not taken, then the NPC (the incremented PC) associated with that branch (used for PC-relative computations) is used to update the PC. If the predicted and actual outcomes are the same, no action is taken, and everything commits normally.

Control instructions limit the independence of computations, even when no data dependecies exist. We implemented a configurable branch predictor that uses one of three schemes:

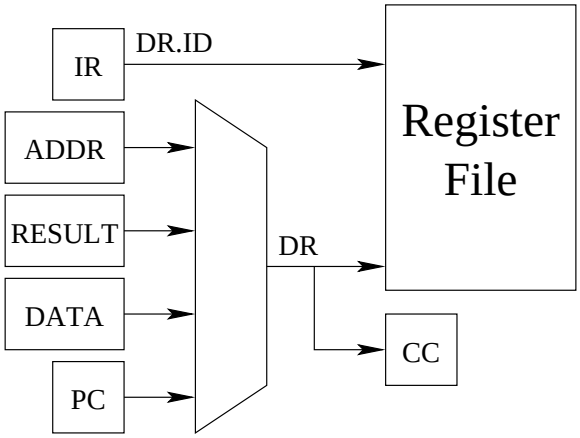


Figure 6: Write-Back (WB) stage of the scalar pipeline design.

always taken, always not taken, based on direction [6]. The last scheme predicts all branches with negative offsets as taken (likely for loops), and positive offsets as not-taken. The choice of which predictor is chosen at simulation startup. When a branch is resolved in the memory stage, a check is made to see if the actual outcome is different than the predicted outcome. When the two differ, a signal is raised (PC.SEL) that tells the IF stage to fetch the branch target on the next cycle. Also, the instructions in IF, ID, and EX are all marked as invalid (squashed). Correctly predicted branches result in no-action taken. Only true branches are predicted. Jump sub-routine instructions are always 'predicted 'as not taken, however they are not used in our benchmarks. We did not consider their performance important for studying how well a superscalar processor performs on our benchmarks. In reality, a small Branch Target Buffer could be used to hold the target addreses of both true branches as well as the jump subroutine instructions.

We feel it is important to mention that our technique of handling mis-predicted branches could easily be applied to handle interrupts and exceptions. The same mechanisms that manage branch misprediction could also be effective at squashing instructions in particular stages of the pipeline as well as transfering control to the handler. However, a detailed examination of the implemention of such matters is not the goal of this paper.

3.2.5 WB: Write-back

The write back stage is fairly simple in our pipelined implementation. It merely chooses the proper result from the various functional units and send that value to the register file (along with the write-enable signals obtained from the control store). Not shown on figure 6 on the diagram is the valid bit, which is present in every stage. The write-enable signals are logically ANDed with the 'valid 'bit to ensure that instructions that were executed along mis-predicted paths do not alter the architectural state of the machine.

4. SUPERSCALAR DESIGN

Our goal was to design a 2 way, in-order processor for the LC3-b ISA. We used our pipeline design as a starting point.

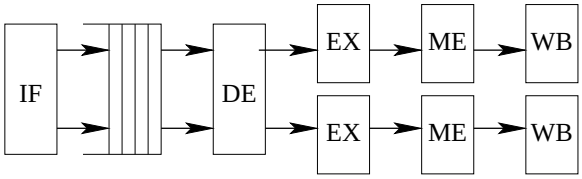


Figure 7: Overview of our 2-way superscalar pipeline design.

4.1 Methodology

Shen's three pipeline idealisms also apply to the design of superscalar pipelined processors. We felt the conclusions we reached for our pipeline design are also valid for a fairly good superscalar design. A 2-way superscalar piplined processor is very much like two scalar pipelines. We also had another motive for keeping the same number of stages : we wanted to directly compare the scalar and superscalar implementations in terms of performance. The next section describes changes and additions for a full-functional 2-way superscalar pipeline.

4.2 Implementation

Our design of scalar pipeline was done with an eye toward a future superscalar pipeline. Our superscalar implementation required a few modifications to the scalar pipeline (aside from a general doubling of components), mainly in the IF and DE stages. Aside from the actual instruction fetch and dependency handling, there is little conceptual difference between the two implementaions. In the following discussion, the 2-way superscalar processor is viewed in terms of two parallel scalar pipelines. The 'first ' of these pipelines is always considered to have the earlier instruction according to program order. This simplies both the discussion and our design. Also please note that the terms 'memory ' and cache are virtually interchangable for this ISA. As stated before, the entire address space of this ISA is small enough to not warrant the uses of caches (assuming modern technology). Alternatively, our zero-latency instruction memory assumption is nearly equivalent to the performance of a good instruction cache on a modern 32 or 64 bit processor.

4.2.1 Instruction Fetch

In order to keep two pipelines busy, the instruction fetch stage must be capable of fetching two instructions per cycle. We could have just have assumed that the instruction memory is 32 bits wide, so that each access fetches two instructions. However, the ISA specifies that instructions must be alighed on word (16 bit) boundaries, but not 32 bit boundaries. We borrowed on the T-logic method[6] described by Shen in his survey of the RS/6000 I-cache. We assume the instruction memory is composed of two banks: one containing even word addresses, one containing odd word addresses. The toggle logic generates the proper index into each of these two banks, and the read values are swapped as necessary (auto-realignment). The result of this is that we can fetch two instructions in one cycle for any valid PC.

With multiple issue processors, there frequently will be stalling associated with data dependencies, e.g. for a pair of fetched instructions, the latter needs to read the result of the former. Issuing only the first instruction on one cycle and only the second instruction on the next cycle is not a viable option for a high-performance processor. We elected to include a 4-instruction buffer inside the IF stage. This allows 2 instructions to be available for decoding on *every* clock cycle. The instruction buffer is controled by a signal from the Decode stage that tells it to advance by one or two instructions for the next clock cycle.

4.2.2 Decode

Two parallel (scalar) pipelines present four times the number of register dependency checks that must be performed compared to our scalar pipeline. Each pipeline's decode stage must look at both pipelines' EX and ME stages for possible dependencies. In addition to the stalling caused by instructions following a load instruction, there is the scenario described above. Stalling is necessary when an instruction in the second pipeline needs the result of the instruction in the first pipeline. No amount of forwarding can prevent this.

Due to the T-logic in the IF stage, if the instruction in the second pipeline must stall, the first pipeline can keep going. Specifically, the decode stage will mark the 2nd instruction as being invalid, and that instruction will get re-issued (on the first pipeline), along with its following instruction, on the next cycle. This still preserves in-order execution *and* allows for maximum performance.

We also doubled the number of read and write ports on the register file. Two instructions can write their results to the register file during the same cycle. If both write to the same register, only the result of the instruction in the 2nd pipeline is used, since it represents the later instruction, according to program order.

4.2.3 Execute

The only modification to the execute stage was to increase the number of inputs for forwarding. For example, the ALU stage may take an operand from any one of 7 places, not counting a possible immediate value from the opcode:

- (1) The value read from the register file (which took place on the previous cycle)
- (2) From the WB stage of either pipeline
- (2) From the ME stage's current address of either pipeline
- (2) From the ME stage's current ALU result of either pipeline

To allow for better synthesis, we decided to use a one-hot encoding for the forwarding signals in the superscalar version. This resulted in a savings of about 200ps, over our initial design that used six 2-to-1 multiplexers. The scalar version uses three 2-to-1 multiplexers to choose from 4 possible inputs. Our choice of encoding has the curious effect of making the minimum cycle time for the superscalar version to be slightly shorter (instead of significantly longer than) that of the scalar pipeline implementation. Otherwise little else is changed; each pipeline contains its own ALU, shifter, and address generating logic.

It is also important to point out that the use of forwarding extends the minumum latency of the EX stage. The critical path is not just through the ALU unit doing a computation, but also through the supporting forwarding logic. However, in the performance section that follows, we show that forwarding and branch prediction still yield considerable performance benefits.

4.2.4 Memory Access

We allow two concurrent memory operations to be performed in a single cycle. We justify this with the fact that the limited number of registers specified by this ISA causes a high level of memory activity for most programs. Furthermore, the small, 16 bit address space makes dual-ported cache/memory quite feasible. Like the register file, concurrent memory operations are handled correctly. If a store and a load instruction execute simultaneously (with the store occuring after the store in program order)and both reference the same address, the load will get the value being stored by the store instruction.

Branches are also resolved in this stage. If two branches are being resolved simultaneously (which is not very improbable), the first branch will have priority. When the first pipeline has a mispredicted branch, it squashes (invalidates) instructions that are in both pipelines' IF, ID, and EX stages. In addition, the instruction residing in the second pipeline's ME stage is also squashed. When a branch in the second pipeline is mispredicted, only the instructions in the IF,ID, and EX stages are squashed.

4.2.5 Write Back

Other than a general doubling of inputs and outputs, the WB stage required little modification. As mentioned above, simultaneous writes to the same register result in only the value of the latter write having effect.

5. VERIFICATION

This is a fairly large scale project. Our source code, all written by hand, for just the superscalar processor is over 100Kbytes in length, which corresponds to over 70 landscape pages, using an 8 pt font. We used several formal techniques to overcome the size and complexity of our code.

We performed serveral methods of testing and verifying the scalar and superscalar processors. Each stage was written, using a verbose naming convention for all signals. At a glance, this made it clear if the right signal was being used somewhere or not. We also adopted standard naming conventions. This and several other techniques served as aides so that the code could be written correctly the first time, with a high probabilty of correctness. It also served to make our code extremely readable and straightforward. Then for each stage, we constructed individual testbenches to ensure the correctness of that stage. Sometimes we constructed testbenches just for the subcomponents of a particular stage. After all the stages had been written, we 'wired' them together, one-by one and tested them out at each step. During this phase, signals that were normally generated by the yet-to-be connected stages were forced to known values. Finally, once everything was complete, we wrote a series of test programs that tried to exploit forseeable problems. These included programs consisting of strings of dependent instructions, unusual branch sequences, small - 2 instruction loops, and various memory access patterns. Each of these programs typically required only a few tens of cycles to complete. This allowed us to visually monitor the progress of every stage on every cycle to verify correctness. Once we were satisfied with the results, we confirmed the correct execution of more complicated programs and benchmarks, by examining the final state of the memory after their execution.

The superscalar processor presented many additional verification challenges. In addition to the extra forwarding paths, the concurrent execution of two instructions provided many more opportunities for failure. It required some creative thinking to create test programs that would exploit potential flaws in our design. A few flaws were uncovered, and all were corrected. Synopsys' design_analyzer software was also used to assist verification. It has a 'check design' feature, which finds unused and cross-connected wires. Our CPU module contains over 100 wire instantiations, so this tool proved to be particularly useful.

The final result of this hard work is two fully functional pipelined processors, one scalar and one superscalar, and a nonpipelined processor, all with seemingly perfect execution.

6. ANALYSIS

6.1 Hardware

Utilizing Synopsys's IC design tools, we successfully synthesized all three target processor implementations using a 0.18 micron process and technical library. Table 4 shows the physical characteristics of our implementations, with the exception of power analysis. Currently, the technical library we are using is not characterized for power analysis; we hope to perform the power analysis of our implementations once this dependence is resolved. As seen in the following table, processor die area increased with each new implementation. This is expected, and is the natural progression: as processor complexity increases, so does the logic required to support it. Specifically, the transition from the basic implementation to the pipelined implementation increased the area 153%, and the transition from pipelined to superscalar resulted in slightly more than doubling (214%) the area again, or an increase of over three times (330%) the area of the basic implementation. There is a paradox, however: the cycle time decreased between the pipelined and superscalar implementations. This reduction is due to an optimization to the forwarding-path circuitry in the superscalar version, giving it a slight (7%) cycle time decrease relative to its pipelined ancestor.

6.2 Software

The LC-3b is largely an educational tool. Hence, there is no high-level language support, such as a C compiler. The ISA, being of load/store type and containing only eight general-purpose registers, forces the programmer to deal with extremely high register pressure. Additionally, the limited instruction set forces programmers to construct missing operations from composite instruction sequences, increasing register pressure even further. Two examples follow:

- op1 || op2 is constructed by: (op1 & op2) ˆ(op1 ôp2)
- op1 * op2 is constructed by mimicking the "long multiplication" algorithm with an iterative sequence of shifts and adds contained within a loop.

Finally, the limited address space further constrains the effective lengths of programs and data.

All of these factors combine to severely constrain benchmarking. Fortunately, there are several "microkernel" benchmarks that lend themselves to easy portability, even within this architecture's constraints. We chose to adapt John McAlpin's STREAM [2] benchmark kernels for our primary benchmarks. Although these kernels are typically used to benchmark the full memory systems of high-performance computers, their structures are useful for examining how static branch prediction and the memory hierarchy of the LC-3b influence its performance. Additionally, we developed a small multiplication benchmark, mul2, to provide a less-predictable counterpoint to STREAM's vectorizable codes.

6.2.1 Adaptation of Stream

Two fundamental changes were required to bring STREAM to the LC-3b: the multiply instruction(s) present in Scale and Triad were replaced with left shift(s) (essentially, a multiplication by a power of two), and the size of the target and source data arrays were reduced to fit within the address space. Although the inputs are reduced, they are very large relative to the LC-3b's address space: for the Copy and Scale kernels, two 8192-word arrays are used (one each for the source and target), for a total of 32KB of data. For the Sum and Triad kernels, three 8192-word arrays are used (two sources and one target), for a total of 48KB of data.

The multiplication benchmark was written explicitly to minimize instruction-level parallelism-this is to provide a counterpoint to STREAM's easily-parallelized codes. This benchmark performs a 16-bit by 16-bit multiplication using an algorithm very similar to the traditional "long multiplication" technique.

6.2.2 Software Scheduling and Loop Unrolling

The STREAM kernels were further optimized by hand, utilizing the principles of software instruction scheduling (to minimize register-register and register-memory dependences) and loop unrolling (to maximize throughput by removing a fraction of the branches). For Copy and Scale, loop unrolling permitted four simultaneous iterations of the loop to occur within the loop's body, permitting a reduction in the loop counter by 4 times. For Sum and Triad, the loop could be unrolled only twice. In both cases, the amount of unrolling is explicitly limited by the number of available registers-a situation that would undoubtedly be improved by adding register renaming to an implementation of the ISA. 'mul2' was intentionally not optimized using either technique.

6.2.3 Special Features

6.2.3.1 Static Branch Prediction

As previously discussed, the pipelined Verilog implementations contain three static branch prediction methods: Taken, Not Taken, and Sign. For the dynamic execution of the benchmarks, there were no differences between Taken and Sign, so only Sign data will be discussed further.

6.2.3.2 Initial Pipeline State

The initial pipeline state of the pipelined Verilog implementations can be configured at run-time. Option '1' fills the pipeline with 'NOP' instructions. Option '0' marks all stages as invalid, and is the configuration used for data collection.

6.2.3.3 Reference Pipeline Simulator

In addition to the Verilog implementations, we developed a stand-alone simulator which fully corresponds to Patt's reference pipelined LC-3b implementation-it was invaluable for demonstrating the effectiveness of our optimized pipeline design. Unfortunately, the comparison cannot be completely accurate since the reference "nominal" pipelined simulator has no branch prediction and must stall until the branch is resolved in the Memory stage.

6.3 Benchmark Results

Table 5 provides the total number of dynamically-executed instructions for each benchmark, as well as the cycle counts for each processor implementation. As is expected, the cycle counts (and execution times) drop as the processors become more sophisticated.

6.3.1 Aggregate Processor Performance

Figures 8 and 9 show the CPI and IPC, respectively, for all compared implementations across all benchmarks. As expected, the CPI drops and the IPC increases as processor complexity increases, for all benchmarks. In particular, figure 8 shows how poorly the Basic implementation performs relative to all subsequent implementations. The Basic implementation also shows a dependence anomaly (its CPI increases abnormally - all other implementations decrease from left to right across the benchmarks) relative to the other implementations on the Sum, Copy, and Scale benchmarks. Figure 9 demonstrates the raw performance improvements of the Superscalar implementation over its pipeline-forwarding antecedent. Table 6 lists the average CPI and IPC per processor for all benchmarks, with all benchmarks equally weighted.

6.3.2 Pipeline Load Balancing

Tables 7 and 8 show the benchmark results of the superscalar implementation, focusing specifically on how static branch prediction affects the load balancing of the pipelines. The Average %Work per Pipeline shows the benchmarks equally weighted, whereas P1 Average Load and P2 Average Load show the load average per pipeline per benchmark. Although P1 always has a higher load, due to the in-order execution of the processor, benchmarks exhibiting low pipeline interdependence (i.e. the STREAM kernels) show remarkably even balance, regardless of the branch predictor. This demonstrates that although the branch prediction method has a significant impact on the execution time (cycles) of each program, the fully-symmetric nature of our superscalar implementation allows significant ILP exploitation. 'mul2 ', the degenerate benchmark case, shows large pipeline interdependence: P1 executes 70-75% of the instructions in the program, depending upon the branch prediction method.

Table 2: Generic sub-computations involved for each instruction in the ISA

Instruction	Subcomputations
Arithmetic	Register Read, ALU OP, Register/CC Write
Logical	Register Read, ALU OP, Register/CC Write
Shift	Register Read, L/R Shift, Register/CC Write
Memory	Register Read, ADD (Address), Memory Access, Register/CC Write
Control	Register/CC Read, ADD (Address), Branch Resolution (Change PC)

Table 3: Latencies for selected subcomputations

Subcomputation	Latency
ALU (ADD,AND,XOR)	486 ps
Variable Shift	337 ps
Register Access	364 ps
Data Memory Access	*variable*

Table 4: Synthesis results of each implementation

Processor	Cycle Time (pico-seconds)	Max Core Speed (MHz)	Die Area (mm^2)	Percent Area Change (over Basic)
Basic	1459.20	685.31	0.0934	-
Scalar Pipeline w/ Fwd	1163.03	859.82	0.1437	153.81
2-way Superscalar	1086.71	920.21	0.3081	214.43

Table 5: Table of Cycle counts for various processor implementations and choices of branch predictors.

Benchmark	Dynamic Instruction Count	Cycle Counts for each processor: Basic	Patt's Pipeline	Our Pipeline (NT) Pred	Our Pipeline (SIGN) Pred.	Superscalar (NT) Pred.	Superscalar (SIGN) Pred
triad	61,447	618,550	102,417	73,733	61,454	45,058	32,780
sum	53,255	553,014	86,033	65,541	53,262	40,963	28,684
copy	24,582	280,622	36,878	30,724	24,589	20,480	12,301
scale	32,774	346,158	45,070	38,916	32,781	24,576	16,397
mul2	6,323	52,653	20,671	12,505	9,316	10,940	7,542

Table 6: Aggregate CPI and IPC Across Implementations

Basic	Patt's Pipeline	Pipeline w/Fwd	Superscalar	
Average CPI	10.04005	1.713139	1.068837	0.583658
Average IPC	0.098511	0.530403	0.913393	1.53112

However, there is little relative difference between the NT and SIGN results for this benchmark. Finally, considering the Average % Work per Pipeline, there is remarkably little difference between branch prediction techniques and our processor's efficiency, in the aggregate.

7. CONCLUSION

The techniques learned in the class for improving processor performance truly work. Additionally, we completely exceeded our performance target of a speedup of 4, between the superscalar and basic implementations: our actual speedups (shown in figure 10), taking into consideration the core speeds, is 20.87 (superscalar to basic) and 11.63 (pipeline with forwarding to basic). Our superscalar implementation has a speedup of 1.79 over the pipeline with forwarding. Finally, our effective MIPS (true MIPS, as measured by equally weighting the benchmarks and taking into consideration the core clock speed), is shown in figure 11.

8. REFERENCES

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (3rd Edition).* Elsevier Science and Technology, 2002.

[2] J. McAlpin. *STREAM.* http://www.cs.virginia.edu/stream.

[3] Y. Patt. *Unpublished manuscript.* Appendix C: The Microarchitecture of the LC-3b, Basic Machine.

[4] Y. Patt. *Unpublished manuscript.* Appendix A: The LC-3b ISA.

[5] Y. Patt. *Unpublished manuscript.* Untitled document on a pipelined LC-3b machine.

[6] J. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors (Beta Edition).* McGraw-Hill, 2003.

[7] E. Smith. *IBM Stretch.* http://www.brouhaha.com/ eric/retrocomputing/ibm/stretch/, 2002.

Table 7: SIGN Branch Prediction

	P1 Retired	P2 Retired	Total Retired	P1 Avg Load	P2 Avg Load	P1 IPC	P2 IPC
triad	32772	28676	61448	53.33%	46.67%	0.9998	0.8748
sum	28677	24579	53256	53.85%	46.15%	0.9998	0.85689
copy	12293	12290	24583	50.01%	49.99%	0.9994	0.99911
scale	16389	16386	32775	50.00%	50.00%	0.9995	0.99933
mul2	4549	1774	6323	71.94%	28.06%	0.6032	0.23522
Average %work	53.08%	46.92%					

Table 8: NT Branch Prediction

	P1 Retired	P2 Retired	Total Retired	P1 Avg Load	P2 Avg Load	P1 IPC	P2 IPC
triad	32772	28675	61447	53.33%	46.67%	0.72733	0.6364
sum	28676	24580	53256	53.85%	46.15%	0.70005	0.60005
copy	14339	10243	24582	58.33%	41.67%	0.70011	0.50012
scale	18435	14339	32774	56.25%	43.75%	0.75009	0.58343
mul2	4759	1564	6323	75.26%	24.74%	0.43501	0.14296
Average %work	55.49%	44.51%					

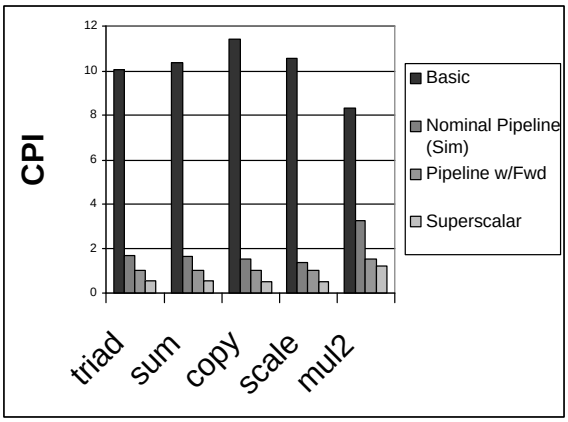


Figure 8: CPI Comparison Across Benchmarks and Implementations

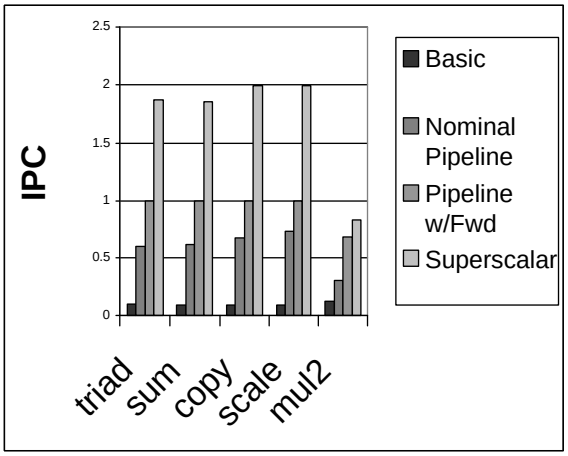


Figure 9: IPC Comparison Across Benchmarks and Implementations

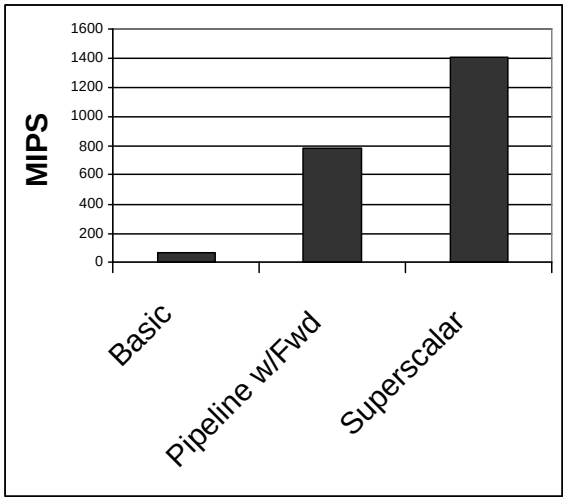


Figure 10: Millions of Instructions Per Second for each Implementaion

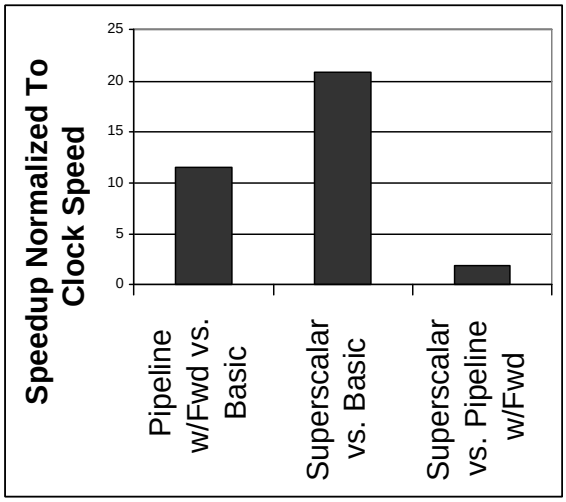


Figure 11: Relative Speedup, with Respect to Clock Speed

APPENDIX

A. BASELINE CODE

Each functional unit (register file, ALU, control store), has its own verilog module.

B. PIPELINE CODE

Our processor is implemented in synthesizable Verilog code. We created two modules for each stage, one with only combinational logic, one with only sequential logic (the stage buffer). The CPU itself is a module that instantiates all the stage modules. The CPU module also contains connections for external memory. A separate 'main ' module instantiates the CPU and memory modules, and provides a textual display of the state of all stages on every cycle, as well as performance counting. Upon startup, memory and register file contents are initialized from text files.

Our pipelined processor of the following verilog modules:

main Generates CLOCK/RESET signals, Monitors the state of all stages, does performance counting, instantiates the CPU and memory modules. Has no external inputs or outputs.

CPU Instantiates all pipeline modules and the wires in-between. External inputs and outputs consist mainly of CLOCK, RESET, and memory signals.

IF_com Combinational logic of the (IF) Instruction Fetch stage. This one is an exception in that it also takes a clock for the actual updating of the PC. The rest of the logic is purely combinational. Branch prediction is also done in this stage.

DE_seq Sequential logic of the (DE) Decode stage. Contains only the stage buffer that exists between the IF and DE stages. Once per every clock cycle, the values outputed by the IF_com module are latched onto the outputs.

DE_com Combinational logic of the (DE) stage. This is responsible for all instruction decoding and dependency checking. The register file is also contained in this stage.

EX_seq Sequential logic of the (EX) Execute stage. This is a buffer that holds the values from the DE stage.

EX_com Combinational logic of the (EX) stage. This includes the ALU, shifter, and address calculation logic.

ME_seq Sequential logic of the (ME) Memory stage, updates on every clock cycle.

ME_com Combinational logic of the ME stage. This module interfaces with the memory module for load/store operations. Condition code reading and branch resolution is also done here.

WB_seq Sequential logic of the (WB) Write-back stage, holds values from ME_com.

WB_com Combinational logic of the (WB) stage. Passes data and control signals to the register file and condition code registers.

C. SUPERSCALAR CODE

The superscalar pipeline code has exactly the same overall structure as the scalar pipeline. However, each module described above contains the inputs and outputs for both pipelines.

D. BASELINE DIAGRAMS

Please See attached pages.

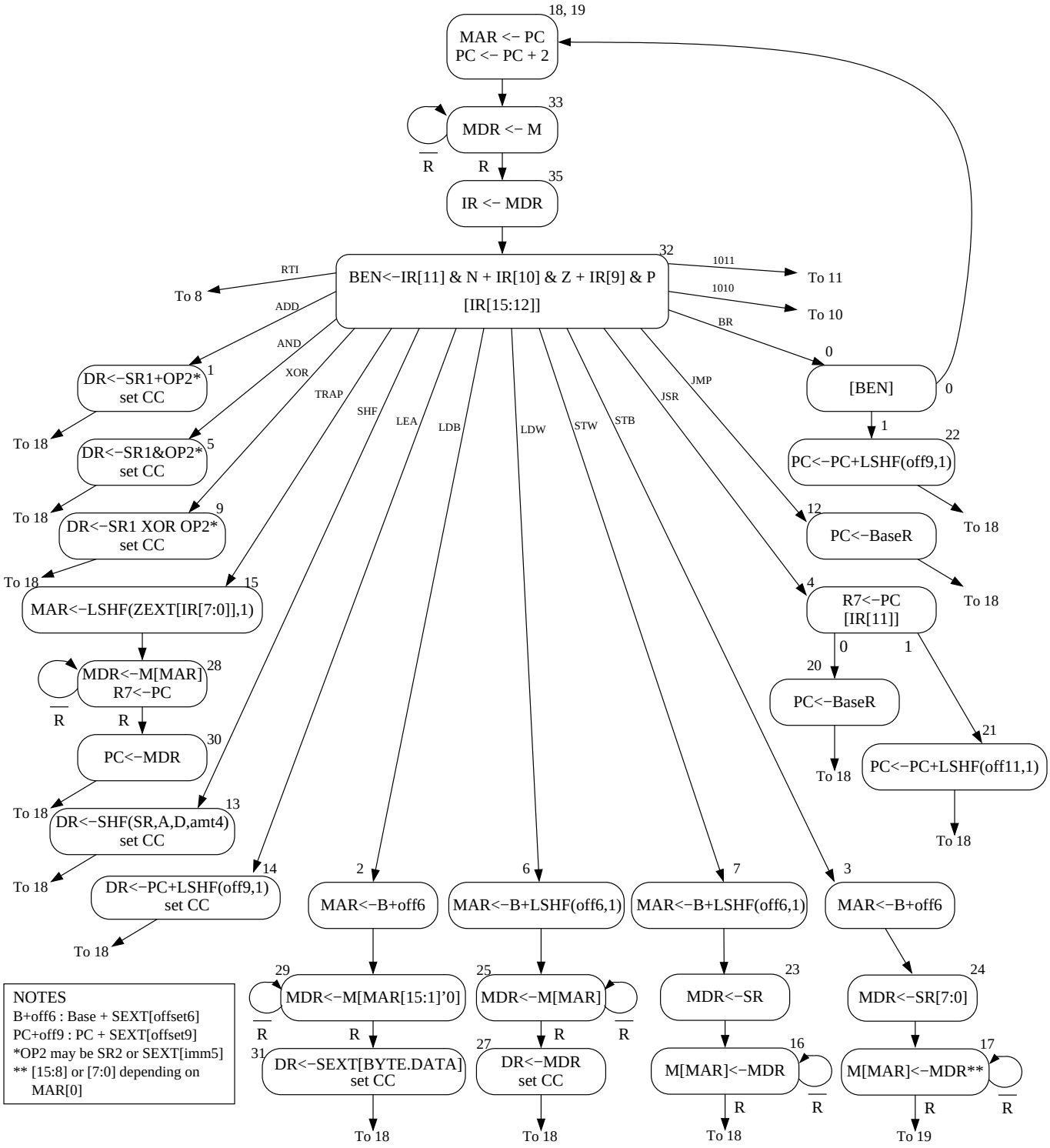


Figure 12: State Diagram of the Basic, non-pipelined implementation, courtesy of Dr. Patt

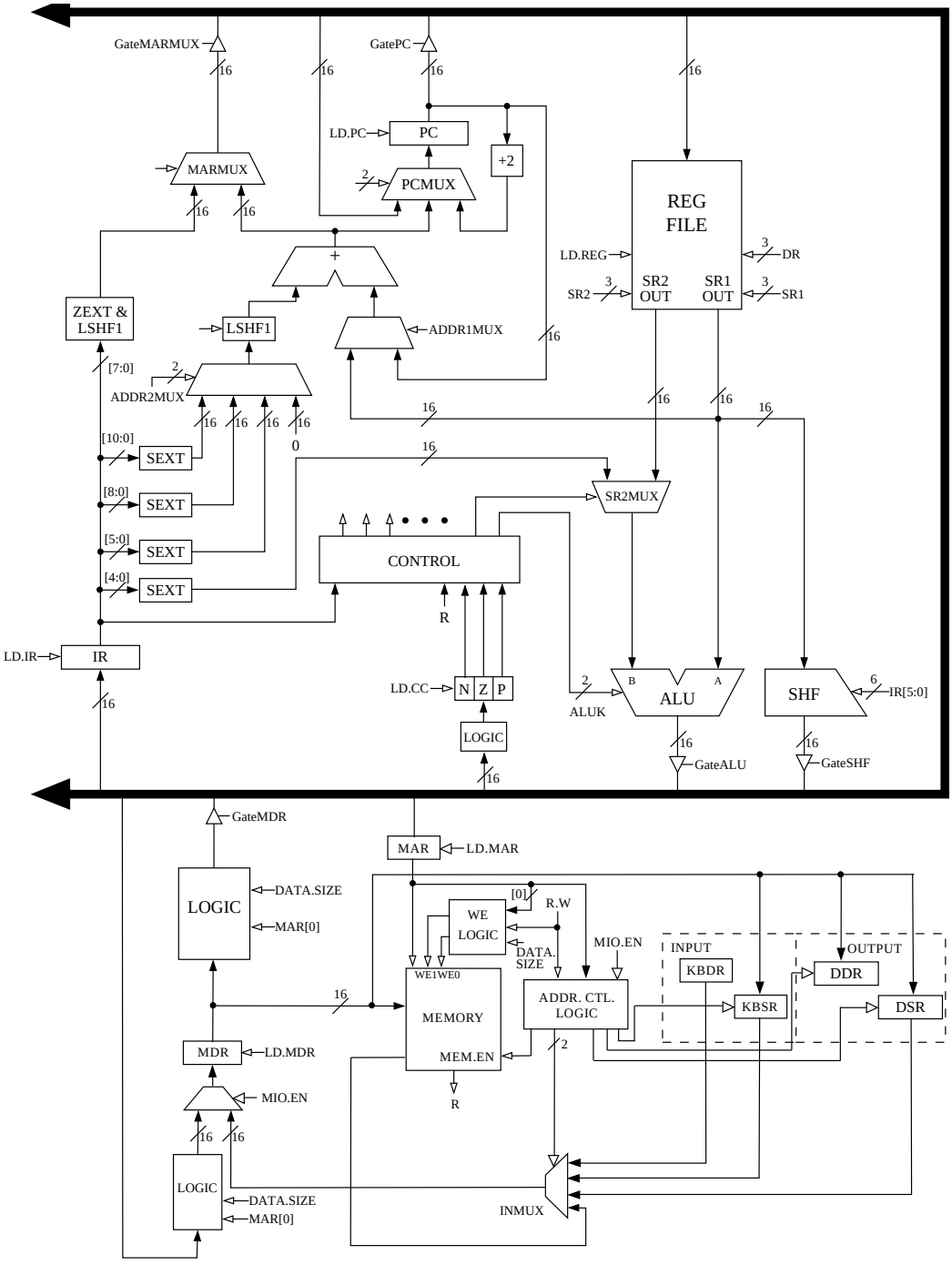


Figure 13: Datapath of the Basic, non-pipelined implementation, courtesy of Dr. Patt