PHILIPPINE ENGINEERING JOURNAL
PEJ 2011; Vol. 32, No. 1: 27- 44

OPTIMIZING A 32-BIT ARM7 MICROPROCESSOR

J.T. Delarmente, L.A.P. Gahol, C.E.L. Paragas, J.R.E. Hizon, and A.P. Ballesil
Electrical and Electronics Engineering, University of the Philippines, Diliman

ABSTRACT

The ARM7 belongs to the Advanced RISC Machines (ARM) family of general-purpose 32-bit microprocessors. Its architecture is based on Reduced Instruction Set Computer (RISC) principles, and employs a three-stage pipeline which results in performance speedup by increasing the microprocessor's throughput.

A previous high-level implementation of the base ARM7 at the Intel Microprocessors Laboratory has a maximum usable clock frequency of 10MHz which is quite slow for a 32-bit processor. The goal of this new implementation was to improve processor speed of the original ARM7 via four techniques, namely: changing the coding style, using one-hot encoding, applying slack borrowing, and adapting architectural modifications.

As with the original, the implementation employed two levels of abstraction, namely Behavioral Level and Register Transfer Level (RTL). The RTL model was synthesized using standard cells on a 0.25µm CMOS process. Synthesis and verification was performed using Cadence Design Systems Software. The performance of this new implementation was evaluated not only for speed but also for area and power consumption. Power measurements were done using Synopsys PrimePower. The maximum clock frequency is 40MHz, area is measured to be 0.99570 mm^2, and observed maximum power is 43.58mW.

1. INTRODUCTION

The ARM7 microprocessor is a RISC machine having a load-store architecture and fixed-length 32-bit instructions having 3-address instruction formats. Its instruction set can be categorized into three main types – data processing, data transfer and control flow instructions. Arithmetic and logical operations, multiplies and PSR transfers are the three main types of data processing instructions while single data transfer, block data transfer and swap fall under data transfer instructions. Control flow instructions, on the other hand, consist of branch instructions and the supervisor call. [1]

The ARM7 processor core employs a simple three-stage pipeline with the following pipe stages: instruction fetch (IF), instruction decode (ID), and execute (EXE). In the IF stage, the instruction is read from the memory and then placed in the Instruction Pipeline Register. The Instruction Decoder then decodes the fetched instruction and the datapath control signals are prepared for the next stage. In the EXE stage, register operands are read. One operand is shifted and then sent to the ALU, while the other is fed directly to the ALU. After performing the desired operation, the result is written back to the destination register. [2]

Correspondence to: Institute of Electrical and Electronics Engineering, University of the Philippines, Diliman

Received: May 16, 2006
Accepted: February 15, 2012

The previous implementation of the ARM7 has a maximum usable clock frequency of 10 MHz and an area of 1.78mm by 1.17mm. Its observed maximum power is 6.115 mW while the observed minimum power is 2.197 mW. [3]

2. DESIGN OPTIMIZATION

In this section, a general overview of the optimization techniques used in the design will be given. The blocks that composed the new ARM7, as well as the optimizations done for each block will also be explained.

2.1. Optimization Techniques

The design of the ARM7 microprocessor made use of four techniques namely: changing the coding style, using one-hot encoding for finite state machines (FSMs), implementing certain architectural changes and using slack borrowing.

The **coding style** used can affect the performance of a design especially when it is translated into its generic components. Thus, different styles were explored to determine the optimal design for speed and/or area. [4] **One-Hot encoding** is implemented by assigning each state to one bit. Its advantage is that it reduces the next-state and output logics for the FSMs. Since only one bit is needed to represent a state, lesser gates are needed and the computation of the signals is faster. The downside here is the increase in area since N states must be encoded in N registers. **Different topologies** for certain blocks were explored and then characterized also in terms of speed and area. This approach made way for a more structural implementation of the design. [5] **Slack-borrowing** is the phenomenon in latch-based systems wherein it is possible for a logic block to utilize time that is left over from a previous logic block. This concept was used in changing the signal timing of the design, producing the final block diagram of the ARM7 microprocessor shown in **Figure 1**, showing the phase transparency of its blocks. Compared to the original design, the produced ARM7 has more components and buses that are transparent at phase 2 rather than at phase 1. Consequently, more control signals are now generated during phase 2.

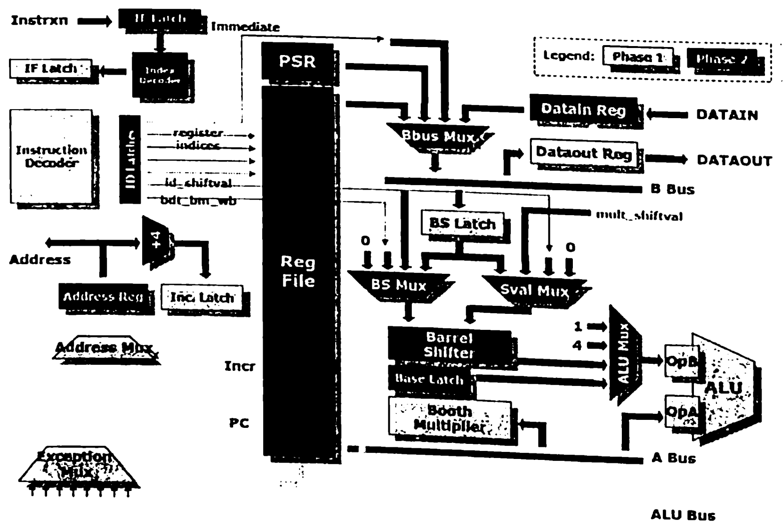

Figure 1. Base ARM7 Block diagram

2.2. Clock and Timing

The ARM7 design is based around 2-phase non-overlapping clocks, which are generated internally from a single input clock signal. This clocking scheme allows the use of level-sensitive transparent latches. Data movement is controlled by passing the data alternately through latches that are open during phase 1, and latches that are open during phase 2. The non-overlapping property of the phase 1 and phase 2 clocks ensures that there are no race conditions in the circuit.

The transparency of the components in relation to different operations done in the execution of an instruction is briefly discussed below:

- Register read operation: The register read buses are updated at the start of phase 2. The read buses send the registers' data through the datapath, and the data arrives at the input of the ALU latches at the end of the phase.
- Shift operation: The 2nd operand passes through the barrel shifter at phase2.
- ALU operation: The ALU has two input latches, both open in phase 1. This allows the ALU to begin processing the operands at phase1 as soon as valid data is available from the ALU latches. They close at the end of phase 1, so that changes in the datapath at phase2 do not affect the output of the ALU. The result is then latched in the destination register at phase 2.

2.3. Component Blocks

The blocks that are introduced first starts with the internal clock generator, then the blocks that comprise the three pipeline stages: the IF stage, the ID stage, and EXE stage. After this, the components of the datapath are discussed.

2.3.1. Clock Generator

For this implementation, a clock generator was used (instead of two input clocks) to produce the two-phase non-overlapping clocks from a single input clock signal *mclk*. It makes use of two cross-coupled NOR gates and several buffers which determine the two clocks' non-overlap time.

2.3.2 IF Block

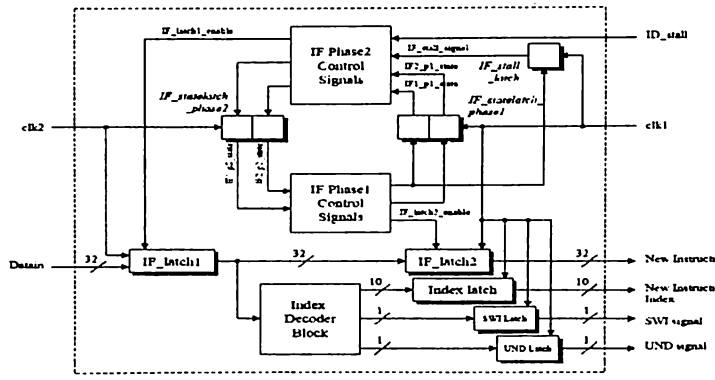

Figure 2. Components of the IF Stage Block

The Instruction Fetch Stage (IF stage) Block shown in **Figure 2**, performs the fetching of instructions from the external memory unit, as specified by the memory address provided by the address register in the ARM7 datapath. The IF stage Block accepts a 32-bit input data from the memory that contains the new instruction, and a 1-bit prefetch abort signal that tells the ARM7 processor if a pre-fetch error has occurred. The IF stage also performs pre-decoding of the instruction type before the instruction is passed to the ID stage. Pre-decoding is performed during the IF stage to lessen the delay incurred in the decoding of the instruction in the Instruction Decoder (ID) Block.

2.3.3. *Index Decoder Block*

The Index Decoder Block shown in the lower part of the IF stage performs the decoding of the type of the instruction from the 32-bit data input. It assigns a 10-bit one-hot code, the *instruction index*, to indicate the instruction type. One-hot encoding is chosen to represent the instruction index because it decreases the complexity of both the encoding of the instruction types and the EXE control unit, making the generation of control signals faster. The instruction index and SWI decoding follow the truth-table shown in **Table 1**, and the undefined instructions are shown in **Table 2**.

Table 1
Index Decoder Block, and SWI Decoder Truth-Table

32-bit ARM7 Instruction bits										SWI signal	Instruction Index [9:0]										
27	26	25	24	23	21	20	16	7	4		9	8	7	6	5	4	3	2	1	0	
0	0	0	1	0	0	0	-	-	0	0	0	0	0	0	0	0	0	0	0	■	MRS
0	0	0	1	0	1	0	0	-	0	0	0	0	0	0	0	0	0	0	■	0	MFRI
0	0	1	1	0	-	0	-	-	-	0	0	0	0	0	0	0	0	0	■	0	MFRI
0	0	0	1	0	1	0	1	-	0	0	0	0	0	0	0	0	0	■	0	0	MSR
0	0	0	0	0	-	0	-	-	0	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	0	0	0	-	1	-	-	0	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	0	0	1	-	0	-	-	0	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	0	0	1	-	1	-	-	0	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	0	1	0	-	1	-	-	0	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	0	1	1	-	0	-	-	0	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	0	1	1	-	1	-	-	0	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	1	0	0	-	0	-	-	-	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	1	0	0	-	1	-	-	-	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	1	0	1	-	0	-	-	-	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	1	0	1	-	1	-	-	-	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	1	1	0	-	1	-	-	-	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	1	1	1	-	0	-	-	-	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	1	1	1	-	1	-	-	-	0	0	0	0	0	0	0	■	0	0	0	DPI_IS
0	0	0	-	-	-	-	-	0	1	0	0	0	0	0	0	■	0	0	0	0	DPRS
0	0	0	0	-	-	-	-	1	1	0	0	0	0	0	■	0	0	0	0	0	MULT
0	0	0	1	-	-	-	-	1	1	0	0	0	0	■	0	0	0	0	0	0	SWAP
0	1	0	-	-	-	-	-	-	-	0	0	0	■	0	0	0	0	0	0	0	SDT
0	1	1	-	-	-	-	-	-	0	0	0	0	■	0	0	0	0	0	0	0	SDT
1	0	0	-	-	-	-	-	-	-	0	0	■	0	0	0	0	0	0	0	0	BDT
1	0	1	-	-	-	-	-	-	-	0	■	0	0	0	0	0	0	0	0	0	DBL
1	1	1	1	-	-	-	-	-	-	■	0	0	0	0	0	0	0	0	0	0	SWI

Table 2
Undefined Decoder Truth-Table

32-bit ARM7 Instruction: bits [27:0]																												Comments
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0	0	1	-	-	-	-	Unused Instrn
0	1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	Undefined
1	1	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Co processor
1	1	1	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Co-processor
0	0	0	0	■	■	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	■	■	1	-	-	-	-	invalid MULT
0	0	0	0	0	0	-	-	■	■	■	■	-	-	-	-	-	-	-	-	1	0	0	1	■	■	■	■	Mult: Rd=Rm
0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0	0	1	■	■	■	■	Mult: Rm=R15
0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	■	■	■	■	1	0	0	1	-	-	-	-	Mult: Rs=R15
0	0	0	0	0	0	-	-	-	-	-	-	■	■	■	■	-	-	-	-	1	0	0	1	-	-	-	-	Mult: Rn=R15
0	0	0	0	0	0	-	-	■	■	■	■	-	-	-	-	-	-	-	-	1	0	0	1	-	-	-	-	Mult: Rd=R15
0	0	0	1	0	-	0	0	■	■	■	■	-	-	-	-	■	■	■	■	■	■	■	0	■	■	■	■	invalid MRS
0	0	0	1	0	-	0	0	1	1	1	1	■	■	■	■	0	0	0	0	0	0	0	0	0	0	0	0	MRS: Rd=R15
0	0	-	1	0	-	1	0	■	■	■	0	■	■	■	■	-	-	-	-	-	-	-	0	-	-	-	-	invalid MFRI
0	0	0	1	0	-	1	0	1	0	0	0	1	1	1	1	■	■	■	■	■	■	■	■	-	-	-	-	invalid MFRI
0	0	0	1	0	-	1	0	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	■	■	■	■	MFRI: Rm=R15
0	0	0	1	0	-	1	0	■	■	■	1	■	■	■	■	■	■	■	■	■	■	■	0	-	-	-	-	invalid MSR
0	0	0	1	0	-	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	■	■	■	■	MSR: Rm=R15
0	0	0	1	■	-	■	■	-	-	-	-	-	-	-	-	■	■	■	■	1	■	■	1	-	-	-	-	invalid Swap
0	0	0	1	0	-	0	0	-	-	-	-	-	-	-	-	0	0	0	0	1	0	0	1	■	■	■	■	Swap: Rm=R15
0	0	0	1	0	-	0	0	-	-	-	-	■	■	■	■	0	0	0	0	1	0	0	1	-	-	-	-	Swap: Rd=R15
0	0	0	1	0	-	0	0	■	■	■	■	-	-	-	-	0	0	0	0	1	0	0	1	-	-	-	-	Swap: Rn=R15
0	1	-	0	-	-	-	-	■	■	■	■	-	-	-	-	-	-	-	-	-	-	-	-	■	■	■	■	SDT-post: Rn=Rm
0	1	-	-	-	-	1	-	■	■	■	■	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	SDT-wb: Rn=R15
0	1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	■	■	■	■	SDT: Rm=R15
1	0	0	-	-	-	-	-	-	-	-	-	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	BDT: No reglist
1	0	0	-	-	-	-	-	■	■	■	■	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	BDT: Rn=R15

2.3.4. *Main Control Unit*

The Main Control Unit is the primary block that controls the operation of the ARM7 microprocessor. It is composed of 6 blocks: the Condition Checker block, the Block Data Transfer Block, the Interrupt Handler Block, the Execute Cycle FSM block, the Phase2 Control Unit Block, and the Phase1 Control Unit Block.

2.3.4.1. *Condition Checker Block*

The function of this block is to check whether the CPSR status flags satisfy the condition provided by the instruction in the *id_cond* field. This block is designed using combinational gates, which is implemented using an if-else statement, and the results of the optimization will be discussed later.

2.3.4.2. *Block Data Transfer (BDT) Block*

Block data transfer instructions consist of store multiple and load multiple instructions. The first cycle is the same for either instruction wherein the memory address offset is calculated. The additional cycles are needed to calculate the next memory address for each transfer by incrementing the current by four. For store multiple instructions, the data from the register file is latched to the memory during phase 1 whereas the data is written to the memory at phase 2. For load multiple instructions, the data from memory is latched during phase 2 and is written to the designated register during phase 2 of the next clock cycle.

The bdt_offset block is composed of 5 levels of adders and a 5-bit 4-input multiplexer. All of the adders except the 5-bit ripple carry adder are used to calculate the number of 1's in the register list, equivalent to the number of registers that will be used for data transfer. The 5-bit ripple carry adder is used to subtract 1 from the total register count. The resulting difference and the total register count are used as some of the inputs in the multiplexer which determines the address offset for the block data transfer instruction. The select signals of the multiplexer are derived from the updown and prepost bits of the instruction opcode. The total register count is also used as one of the inputs in the BS multiplexer when a base register write-back is required.

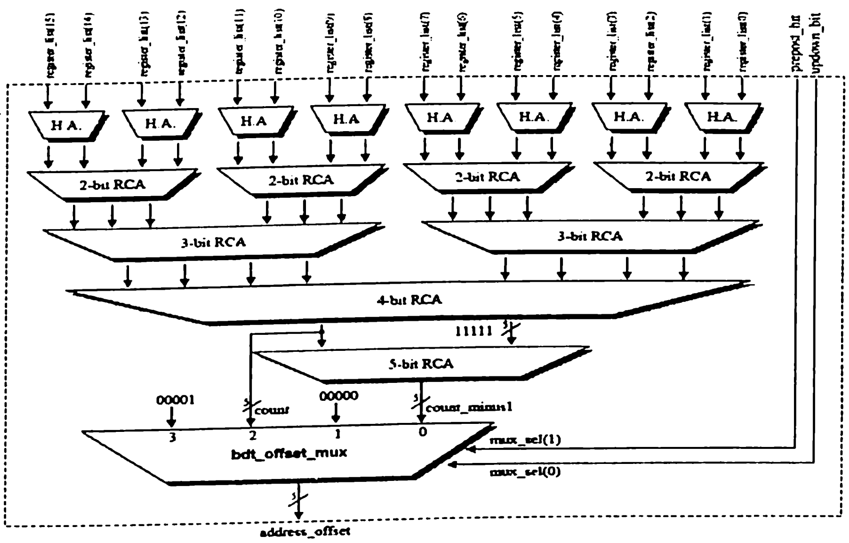

Figure 3. Block Data Transfer Offset Block

There are several differences between the old implementation and the new one. The old implementation uses a for-loop to calculate the total register count and a to_vector function to convert it into binary form. It can be seen that the newer implementation is simpler and more structural than the old one. Note that the conditions for the multiplexer used in the new implementation are still the same as that of the old one. For simplification, the BS multiplexer and immediate multiplexer in the execute stage controller produce shifted versions of the base write-back value and address offset respectively. The shift done is equivalent to multiplying the input by four, thus the shift value of the barrel shifter is fixed to zero for a block data transfer instruction.

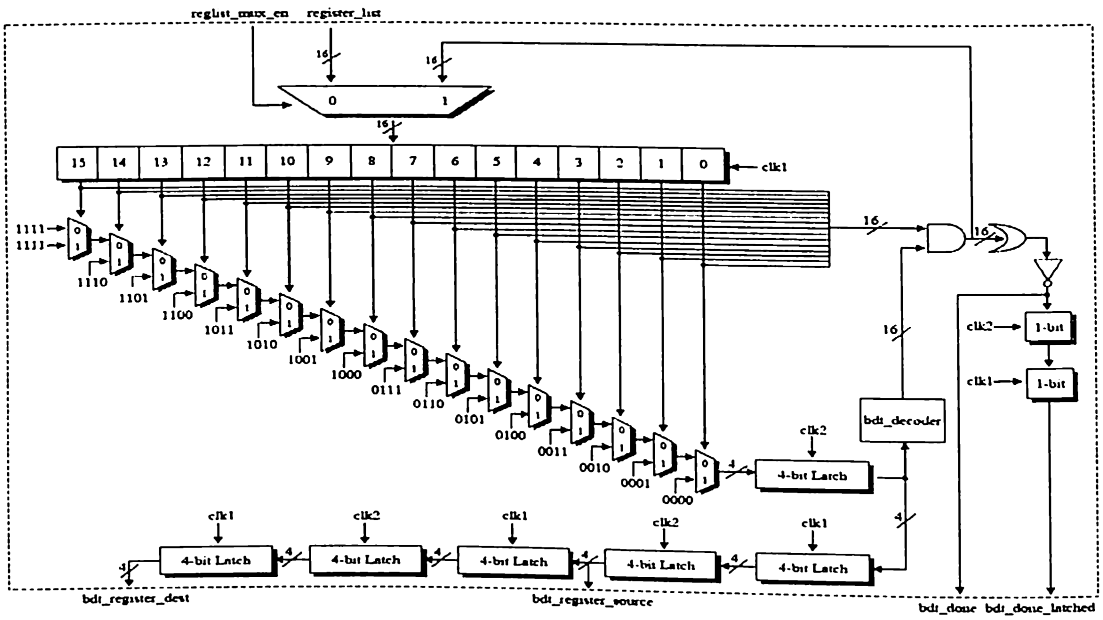

Figure 4. Block Data Transfer Block

The bdt_block is composed of 16 4-bit 2-input multiplexers, a 16-bit 2-input multiplexer, a 16-bit latch, 6 4-bit latches, 2 1-bit latches, a 4x16 decoder, and some logic blocks. It determines the source or destination registers needed for transfer. The register priority used in the old implementation is still applied to the new one. This priority scheme is implemented by the 16 4-bit 2-input multiplexers and the 4x16 decoder which determine which register is next for data transfer. The 4-bit latches are used to indicate the current register for transfer at the correct cycle depending on whether it's a load or a store operation. The 1-bit latches and logic blocks are used to determine when the block data transfer operation is done. Compared to the old implementation where there is a unique cycle state for each single transfer depending on which registers are included in the register list, the new implementation locks the cycle state to a fixed value. This fixed cycle state is only 'unlocked' and thus will go to a new state when the decoding logic indicates that the block data transfer instruction is finished.

2.3.4.3. Interrupt Handler Block

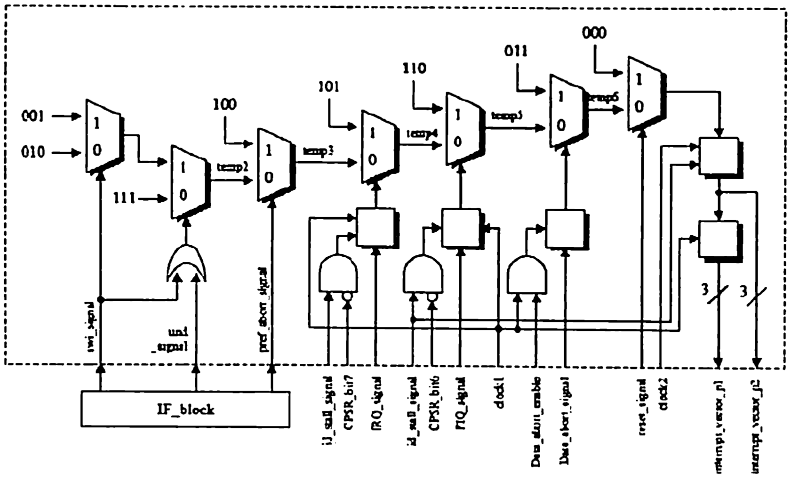

Figure 5. Interrupt Handler Block

The Interrupt Handler Block shown in **Figure 5** processes the interrupts detected by the ARM7 microprocessor, using a priority list provided in its datasheet. The final interrupt mode is encoded in a 3-bit binary code. Two sets of interrupt vectors are then stored, the first interrupt vector is latched during phase2, and the output of this latch is then stored during phase1. These are sent to the control unit to be used in performing the necessary interrupt routines.

Here, the interrupt handler block is not fully asynchronous, although most of the interrupt signals (like FIQ and IRQ) are. It assumes that most of the interrupts can be detected during phase1. Only the external reset is treated as an asynchronous signal. The interrupt handling routine is similar for the FIQ, IRQ, SWI, undefined, and pre-fetch abort interrupts. Their routines follow a three-cycle operation: In the first cycle, the contents of the PC are saved into the register R14 of the interrupt mode, and the CPSR is saved into the SPSR of the interrupt mode. Also, the next address is forced to the exception address. At the second cycle, the address stored in R14 is modified into the correct return address, and the instruction at the exception address is fetched. At the last cycle, this instruction is decoded and is passed to the control unit to be executed in the next cycle. The difference in the reset interrupt routine is the continuous fetching of instructions while the external reset is asserted. When de-asserted, it resumes a three-cycle operation similar to the one mentioned above.

The difference in the handling of data abort interrupts depends on the instruction. For a swap, the instruction is immediately aborted. For a single data transfer, the instruction will also be aborted immediately, but the base register will be modified when write-back is enabled. For a block data transfer, the instruction will resume until it is finished, but register writing is prevented. After these, it does a four-cycle operation. In the 1st cycle, the contents of the base latch (the original data from the base register) are transferred back to the base register. The next three cycles then follow the same process similar to the other interrupts.

These changes in the interrupt handling routine that were added to correct the previous implementation, and the separation of the interrupt detector as a new block, are the major changes from the previous implementation.

2.3.4.4. Execute Cycle FSM Block

The EXE Cycle FSM Block contains the finite state machine that controls the operation of the datapath during each clock cycle. The states are encoded in a one-hot-code to reduce the complexity in the generation of the datapath control signals.

Table 3
Execute Cycle FSM Truth-Table

reset	cmd_passed	no_interrupt_check	no_user_data_abt_check	data_abort_check	Instruction Index	imm_bit	prepost_idx_bit	updown_bit	bw_psr_bit	wb_src_bit	loadstore_set_bit	data_abort_signal	dest_pc_signal	pc_in_reglist_signal	bit_done_signal(tune)	bit_done_latched(bus)	mult_done_signal	Current EXE Cycle	EXE6	EXE5	EXE4	EXE3	EXE2	EXE1
						Instruction bits [25:20]													Next EXE_cycle [5:0]					
						25	24	23	22	21	20								5	4	3	2	1	0
1	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	0	0	0	1
0	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	EXE0	0	0	0	0	0	1
0	0	1	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	0	0	0	1
0	1	0	0	1	-	-	-	-	-	-	-	-	-	-	-	-	-	EXE1	0	0	0	0	1	0
0	1	0	0	1	-	-	-	-	-	-	-	-	-	-	-	-	-	EXE2	0	0	0	1	0	0
0	1	0	1	0	-	-	-	-	-	-	-	-	-	-	-	-	-	EXE1	0	0	0	1	0	0
0	1	0	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	EXE3	0	0	1	0	0	0
0	1	0	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	EXE4	0	0	0	0	0	1
0	1	1	0	0	BBL	-	-	-	-	-	-	-	-	-	-	-	-	EXE1	0	0	0	0	1	0
0	1	1	0	0	BBL	-	-	-	-	-	-	-	-	-	-	-	-	EXE2	0	0	0	1	0	0
0	1	1	0	0	BBL	-	-	-	-	-	-	-	-	-	-	-	-	EXE3	0	0	0	0	0	1
0	1	1	0	0	MRS	-	-	-	-	-	-	-	-	-	-	-	-	EXE1	0	0	0	0	0	1
0	1	1	0	0	MFRI	-	-	-	-	-	-	-	-	-	-	-	-	EXE1	0	0	0	0	0	1
0	1	1	0	0	MSR	-	-	-	-	-	-	-	-	-	-	-	-	EXE1	0	0	0	0	0	1
0	1	1	0	0	BDT	-	-	-	-	-	1	-	-	-	-	-	-	EXE1	0	0	0	0	1	0
0	1	1	0	0	BDT	-	-	-	-	-	0	-	-	-	0	-	-	EXE1	0	0	1	0	0	0
0	1	1	0	0	BDT	-	-	-	-	-	0	-	-	-	1	-	-	EXE1	0	0	0	1	0	0
0	1	1	0	0	BDT	-	-	-	-	-	0	-	-	-	0	-	-	EXE2	0	0	0	0	1	0
0	1	1	0	0	BDT	-	-	-	-	-	0	-	-	-	1	-	-	EXE2	0	0	0	1	0	0
0	1	1	0	0	BDT	-	-	-	-	-	1	-	-	-	-	0	-	EXE2	0	0	0	1	0	0
0	1	1	0	0	BDT	-	-	-	-	-	1	-	-	-	-	1	-	EXE2	0	0	1	0	0	0
0	1	1	0	0	BDT	-	-	-	-	-	0	-	-	-	-	-	-	EXE3	0	0	0	0	0	1
0	1	1	0	0	BDT	-	-	-	-	-	1	-	-	-	-	0	-	EXE3	0	0	0	1	0	0
0	1	1	0	0	BDT	-	-	-	-	-	1	-	-	-	-	1	-	EXE3	0	0	1	0	0	0
0	1	1	0	0	BDT	-	-	-	-	-	0	-	-	-	-	-	-	EXE4	0	0	0	0	1	0
0	1	1	0	0	BDT	-	-	-	-	-	1	1	-	-	-	-	-	EXE4	0	0	0	0	0	1
0	1	1	0	0	BDT	-	-	-	-	-	1	0	-	0	-	-	-	EXE4	0	0	0	0	0	1
0	1	1	0	0	BDT	-	-	-	-	-	-	0	-	1	-	-	-	EXE4	0	1	0	0	0	0
0	1	1	0	0	BDT	-	-	-	-	-	-	-	-	-	-	-	-	EXE5	1	0	0	0	0	0
0	1	1	0	0	BDT	-	-	-	-	-	-	-	-	-	-	-	-	EXE6	0	0	0	0	0	1
0	1	1	0	0	DPI_IS	-	-	-	-	-	-	-	1	-	-	-	-	EXE1	0	0	0	0	1	0
0	1	1	0	0	DPI_IS	-	-	-	-	-	-	-	0	-	-	-	-	EXE1	0	0	0	0	0	1
0	1	1	0	0	DPI_IS	-	-	-	-	-	-	-	-	-	-	-	-	EXE2	0	0	0	1	0	0
0	1	1	0	0	DPI_IS	-	-	-	-	-	-	-	-	-	-	-	-	EXE3	0	0	0	0	0	1
0	1	1	0	0	DPRS	-	-	-	-	-	-	-	-	-	-	-	-	EXE1	0	0	0	0	1	0
0	1	1	0	0	DPRS	-	-	-	-	-	-	-	0	-	-	-	-	EXE2	0	0	0	0	0	1
0	1	1	0	0	DPRS	-	-	-	-	-	-	-	1	-	-	-	-	EXE2	0	0	0	1	0	0
0	1	1	0	0	DPRS	-	-	-	-	-	-	-	-	-	-	-	-	EXE3	0	0	1	0	0	0
0	1	1	0	0	DPRS	-	-	-	-	-	-	-	-	-	-	-	-	EXE4	0	0	0	0	0	1
0	1	1	0	0	MULT	-	-	-	-	-	-	-	-	-	-	-	-	EXE1	0	0	0	0	1	0
0	1	1	0	0	MULT	-	-	-	-	-	-	-	-	-	-	-	0	EXE2	0	0	0	0	1	0
0	1	1	0	0	MULT	-	-	-	-	-	-	-	-	-	-	-	1	EXE2	0	0	0	0	0	1
0	1	1	0	0	SWAP	-	-	-	-	-	-	-	-	-	-	-	-	EXE1	0	0	0	0	1	0
0	1	1	0	0	SWAP	-	-	-	-	-	-	-	-	-	-	-	-	EXE2	0	0	0	1	0	0
0	1	1	0	0	SWAP	-	-	-	-	-	-	0	-	-	-	-	-	EXE3	0	0	1	0	0	0
0	1	1	0	0	SWAP	-	-	-	-	-	-	1	-	-	-	-	-	EXE3	0	0	0	0	0	1
0	1	1	0	0	SWAP	-	-	-	-	-	-	-	-	-	-	-	-	EXE4	0	0	0	0	0	1
0	1	1	0	0	SDT	-	-	-	-	-	-	-	-	-	-	-	-	EXE1	0	0	0	0	1	0
0	1	1	0	0	SDT	-	-	-	-	-	0	-	-	-	-	-	-	EXE2	0	0	0	0	0	1
0	1	1	0	0	SDT	-	-	-	-	-	1	-	-	-	-	-	-	EXE2	0	0	0	1	0	0
0	1	1	0	0	SDT	-	-	-	-	-	-	-	0	-	-	-	-	EXE3	0	0	0	0	0	1
0	1	1	0	0	SDT	-	-	-	-	-	-	0	1	-	-	-	-	EXE3	0	0	1	0	0	0
0	1	1	0	0	SDT	-	-	-	-	-	-	1	1	-	-	-	-	EXE3	0	0	0	0	0	1
0	1	1	0	0	SDT	-	-	-	-	-	-	-	-	-	-	-	-	EXE4	0	1	0	0	0	0
0	1	1	0	0	SDT	-	-	-	-	-	-	-	-	-	-	-	-	EXE5	0	0	0	0	0	1
Otherwise																			0	0	0	0	0	0

2.3.4.5. Phase1 Control Unit Block

For the phase1 control unit block, the output signals are sent every phase1, and remain valid for one cycle. Discussed below are the output signals generated by the phase1 control unit block and their usages.

Table 4
Output Signals of the Phase 1 Control Unit Block

Output signals	Description
if_flush	'Flushes' or resets the IF latches. This happens whenever a system reset or an interrupt occurs, or a branch instruction.
int_flush	Resets the latches within the interrupt handler block after the necessary interrupt routine finishes execution.
abort_latch_enable	Enables the abort latch within the interrupt handler.
int_disabler_signal	Disables FIQs and IRQs in the **CPSR** during reset.
vbit_in	Enables the storing of the overflow (V) bit into the **CPSR** flags.
storedata_tocpsr	Enables the storing of a 32-bit data into the **CPSR**.
storedata_tospsr	Enables the storing of a 32-bit data into the active **SPSR**.
storeflag_tocpsr	Enables the storing of only the flag bits, or the 4 most significant bits in the 32-bit input, into the **CPSR**.
storeflag_tospsr	Enables the storing of only the flag bits, or the 4 most significant bits in the 32-bit input, into the active **SPSR**
change_mode	Enables the storing of the new operating mode into the mode bits of the **CPSR**.
cpsr_set	Enables the setting of the **CPSR** flags using the flags sent by the ALU block.
inputmux_ tocpsr	Selects the input multiplexer before the **CPSR**, which selects what type of input passes into the CPSR: the 32-bit input of the **PSR** block, or the 32-bit data from the active **SPSR**.
inputmux_tospsr	Selects the input multiplexer before the SPSR, which selects what type of input passes into the **SPSR**: the 32-bit input of the PSR block, or the 32-bit data from the **CPSR**.
output_select	Selects the output of the **PSR** block: the data within the **CPSR**, or the data stored in the active **SPSR**.
index_dest_ mux_sel	Selects the destination register for the current instruction cycle. 00 – id_regopRd_p1 01 – "1110" 10 – id_regopRn_p1 11 – bdt_register_dest_signal
Reg_mode_ mux_sel	Determines which mode the register should be in. 00 – psr_mode_bits 01 – "10011" or "11011" or "10111" or "10010" or "10001" or "10000" (depends on interrupt vector) 10 – "10000" 11 – "10000"
we_reg	Indicates if the register is to be written or not.
we_PC	Indicates if a write is to be done on the PC.
PCin_sel	Selects from which block the PC will get its value. The inputs are either from the incrementer or from the ALU block.
datain_reg_in	Enables the Data in Register to accept the data present in the external data bus. This signal is gated with the phase2 clock.
base_latch_in	Enables the Base Latch to accept the data present in the ALU bus. This signal is gated with the phase2 clock.
alu_opcode_ mux_sel	Selects the ALU opcode that determines the operation that will be done in the ALU block. 00 – "1101" 01 – "0100" 10 – "0010" 11 – id_alu_opcode_p1 or booth_logic
address_mux_sel	Selects which address should be sent to the address register. 00 – data in incrementer latch 01 – data in PC register 10 – result of ALU block 11 – output of exception mux
address_reg_in	Enables the address register to store its 32-bit input coming from the address multiplexer. This signal is gated with the phase2 clock.
nmreq_signal	Output signal of the ARM7 core. It tells the external memory managemenf unit that a memory access is required in the next phase. It is low-asserted.
nrw_signal	Output signal of the ARM7 core. It tells the external memory management unit that a read or write memory operation is required in the next phase. It is low-asserted.
nbw_signal	Output signal of the ARM7 core. It tells the external memory management unit that a byte or word memory transfer operation is required in the next phase. It is low-asserted.

2.3.4.6. Phase2 Control Unit Block

For the phase2 control unit block, the output signals are generated at phase2, and remain valid for one cycle. Listed below are the output signals generated by this block, and their usages.

Table 5
Output Signals of the Phase 2 Control Unit Block

Output signals	**Description**
booth_mux_select	Signals the booth multiplier block to accept the multiplier coming from the A bus.
reglist_mux_en	Selects from which component the BDT block should use as its input. The BDT block may get its input either from the latched register list from the instruction bit code or from the decoded register list the BDT block generated.
mode_out_sel	Used for post-indexed single data transfers during privileged mode, when the write-back bit option is set. It forces non-privileged mode during the data transfer.
index_opA_mux_sel	Used to select the multiplexer that outputs the operandA source register index for the register file. 00 – id_regopRn_p2 or id_regopRd_p2 01 – id_regopRm_p2 or id_regopRs_p2 10 – "1111" 11 – "1110"
index_opB_mux_sel	Used to select the multiplexer that outputs the operandB source register index for the register file. 00 – bdt_register_source_signal or id_regopRm_p2 01 – id_regopRn_p2 10 – id_regopRd_p2 11 – id_regopRs_p2
dataoutreg_in	Enables the Dataout Register to accept and store its 32-bit input. This signal is gated with the phase1 clock.
loadbyte_enable	Enables a latch in the load byte block to accept the last 2 bits of the computed effective address for a load-byte operation. This is also gated with the phase1 clock.
Bbit_mux_sel	Used to select the multiplexer in the load byte block. 0 – '0' 1 – id_bw_psr_bit_p2_signal
Bbus_mux_sel	Used to select the B-bus multiplexer. 00 – Operand2 from the register file 01 – PSR 32-bit output 10 – Immediate value from ID 11 – Data from Datain register
BS_mux_sel	Used to select the BS multiplexer. 00 – "0000_0000"h 01 – "000_0000"h & bdt_b_wb[4:0] & "00" 10 – data from B-bus 11 – data from BS latch
sval_mux_sel	Used to select the sval multiplexer. 00 – "00000000" 01 – shift value form booth multiplier 10 – shift value from ID 11 – shift value form BS latch
ALU_mux_sel	Used to select the ALU multiplexer. 00 – output of Barrel Shifter 01 – "0000_0004"h 10 – "0000_0000"h 11 – output of Base Latch
shifttype_mux_sel	Used to select the multiplexer which outputs the ff: 0 – "00" 1 – id_shifttype
BS_latch_in	Enables the BS latch to accept the data present in the B-bus. This signal is gated with the phase1 clock.

2.3.5. Datapath optimizations

The ARM7 datapath consists of the Barrel Shifter, the Booth Multiplier Block, the ALU block, the Register File, and several latches and multiplexers.

2.3.5.1. Barrel Shifter

The shifter was designed by using cascaded multiplexers, both for the data and the carryout. The ARM7 is able to perform a maximum of 8-bit shifts for four types of shifts, namely Logical Shift Left (LSL), Logical Shift Right (LSR), Arithmetic Shift Right (ASR) and Rotate Right (ROR). **Figure 6** shows the block diagram of the cascaded multiplexers used for the LSL operation. As an example, for a required shift value of 21, multiplexers 1, 3 and 5 would produce values shifted by 1, 4 and 16 respectively, producing a total shift value of 21. The bits of the shift values are directly used as the select signals for the muxes, eliminating the supposed need for a decoding logic in the shifter. For the carryout, 9 muxes were cascaded instead of 8. This is in order to account for the special cases in the carryout as required by ARM7.

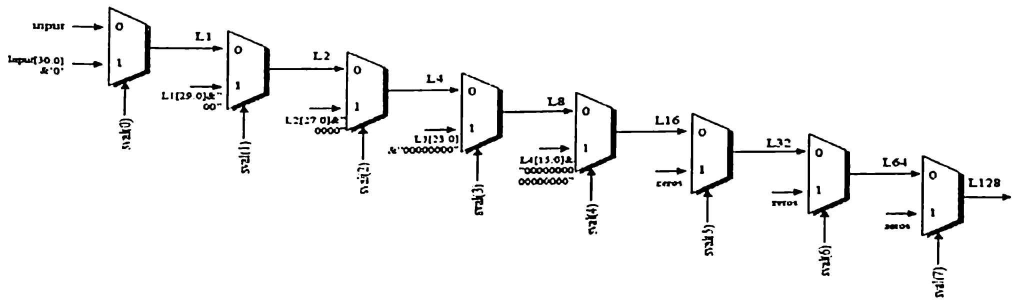

Figure 6. Shifter Components for Logical Shift Left (LSL)

The same principle of cascaded multiplexers is also used for the three other shift types. For ROR however, an extra multiplexer was needed to select the final output. This was in order to account for the special case of Rotate Right Extend (RRX, denoted by ROR0). Also, another extra multiplexer is used to select the final carryout value for ROR32, which is another special case for rotate. At the end of these cascaded multiplexers is a final mux whose select signal is the shift type. This selects the final output depending on the desired shift type.

2.3.5.2. Multiplier Block

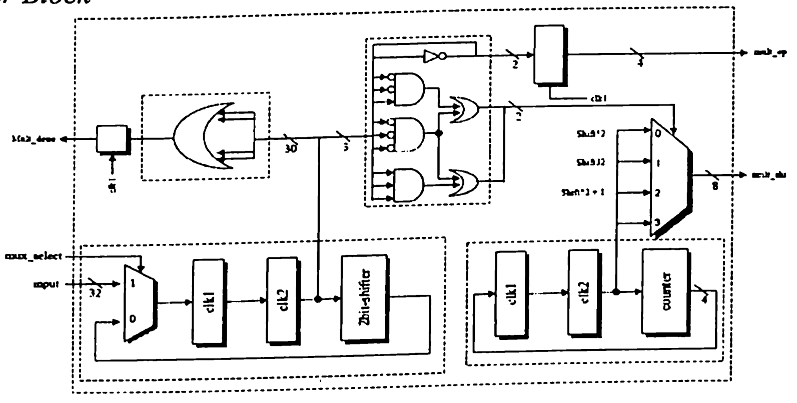

Figure 7. Multiplier Block

The multiplier block accepts a 32-bit *input* from the Abus (the multiplier) and a *mux_select* control signal from the control unit. The mux_select enables the mux to accept the multiplier from the Abus, and also serves as the reset of the latch succeeding it. This block sends out signals to other parts of the datapath depending on the resulting encoding as dictated by the Modified Booth Algorithm. It has three parts, namely, the counter which keeps track of the number of partial products, the mult_done block which determines if early termination of the instruction is to take place, and the booth_block which shifts the multiplier two bits to the right, reads its 3 LSBs then uses combinational logic to select the appropriate shift value.

For the booth_encoder, combinational logic is used in order to create the select signal for the multiplexer and the *mult_opcode*. The mux chooses among the three possible shift values that the multiplicand can assume. For a multiplicand of 0xM, *mult_shiftval* should be 32 as to produce a zero. For ±1xM, *mult_shiftval* assumes a value of two times the count value, in accordance to the two shifts as dictated by the modified Booth algorithm. Lastly, for ±2xM, *mult_shiftval* takes on a value of twice the count value plus 1, where plus 1 accounts for the required multiplication by 2. The encoding used for this block is shown in **Table 6.**

Table 6
Booth Encoding

Multiplier Bit-Pair	Bit on Right	Multiplicand	opcode_slice	sval_sel
00	0	0 x M	10	01
00	1	+1 x M	10	00
01	0	+1 x M	10	00
01	1	+2 x M	10	10
10	0	-2 x M	01	10
10	1	-1 x M	01	00
11	0	-1 x M	01	00
11	1	0 x M	01	01

Originally, the goal was to remove the counter in the multiply block. However, since the number of cycles in the multiply block have to be tracked, this was not done. (The control unit only allows for 2 states in the multiply instruction, hence there is no way of knowing the number of cycles that has taken place.) The original counter which made use of the plus operator was replaced with a combinational counter. This can be done with the number of bits being fixed. This resulted to a small reduction in area.

2.3.5.3. Register File

The optimizations performed on the register file are: the removal of the demultiplexer in the original implementation, which is placed between the 32-bit input and the latches and is controlled by the destination index, since this is not needed when all the latches are gate-enabled, and the recoding of the registers' indices decoders to get the fastest implementation.

2.3.5.4. PSR Block

The major change in the PSR Block is the reorganization of the design to get a more structured look. In the original implementation, the CPSR and SPSR blocks are coded in a behavioral manner. In this design, the component bits of the CPSR and SPSR are broken down into different latches, shown in **Figure 8** and **Figure 9** respectively. Note that the CPSR and SPSR blocks are composed of a block of latches whose outputs are appended to produce a 32-bit output.

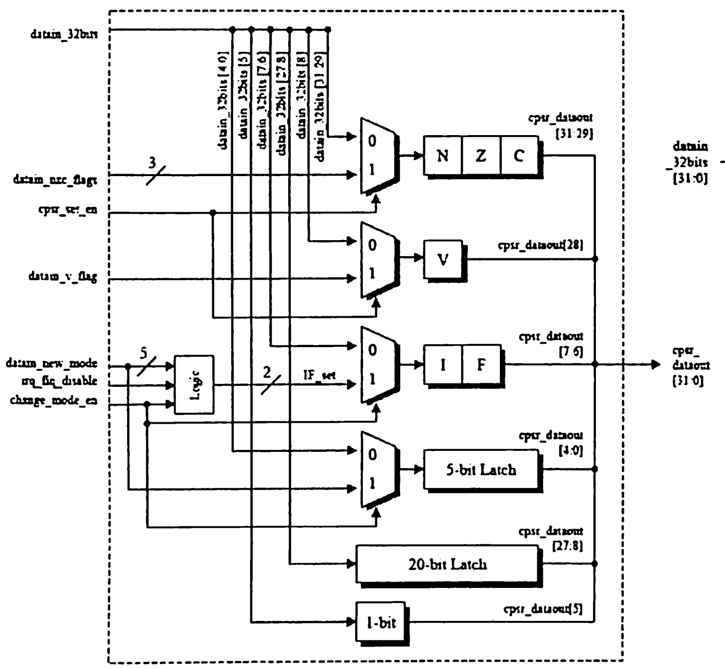

Figure 8. Structure of the CPSR Block

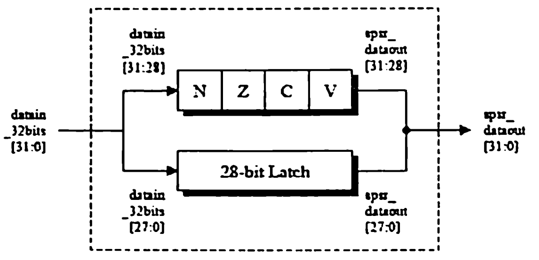

Figure 9. Structure of the SPSR Block

2.3.5.5. ALU Block

The main modification made in ARM's ALU is changing the adder from a 32-bit Carry-Select to a 32-bit Parallel Prefix Kogge-Stone Tree Adder. This adder makes use of Lookahead Logic, making it significantly faster than most adders. This increase however, comes at the expense of area since it occupies an area almost two times than the CSA.

3. METHODOLOGY

The design flowchart starts with the design code, implemented using VHDL syntax. The code is compiled and tested using the software NCLaunch from the Cadence Design System Software. The verified code is then synthesized using the Ambit BuildGates tool, where a gate-level design is generated. This tool produces the VHDL netlist, the GCF file, and the SDF file, taking into account the standard cell library (which specifies the format of each cell) and the clocking constraints (set by the user). After synthesis, simulation is redone to test the functionality of the generated netlist, and to check the timing parameters of the generated schematic. Synthesis is repeated in case timing specifications are not met.

Once the results of the synthesis are satisfactory, the generation of the physical layout is performed using the Silicon Ensemble Ultra (SE-Ultra) tool. SE uses the standard cell library to automatically construct a layout from the netlist generated by Ambit. The metal layers, cell sizes, and other pertinent data for layout are specified in the standard cell libraries. NCLaunch is again used to verify that the layout is functioning, since the netlist generated from SE takes into account the delays introduced by routing.

Last in the flow is power estimation, and this done through the use of Synopsys' PrimePower. This tool provides a detailed account of the power consumption per component block, and gives a breakdown of the total power into its components, namely leakage and dynamic power. Furthermore, dynamic power is broken down into its subtypes given by switching, internal, x-tran and glitch. With speed being the primary concern of this project, power is only measured at the end of the design flow. However, when power is of the utmost concern, the designer should go back to coding, synthesis or place and route if power constraints are not met.

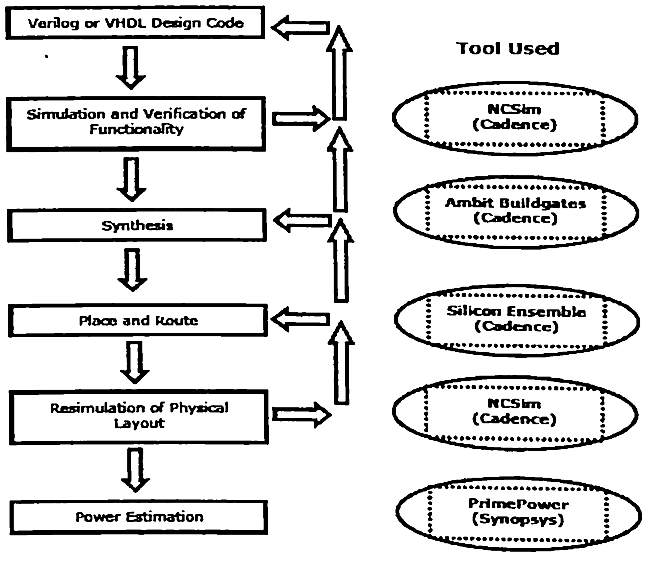
Verilog or VHDL Design Code
Simulation and Verification of Functionality
Synthesis
Place and Route
Resimulation of Physical Layout
Power Estimation
Tool Used
NCSim (Cadence)
Ambit Buildgates (Cadence)
Silicon Ensemble (Cadence)
NCSim (Cadence)
PrimePower (Synopsys)

Figure 10. Project Flow

4. RESULTS

From the four design optimization techniques discussed briefly in Section 2.1, only the effects of change of coding style and architectural changes were quantified. Given the original ARM7 and with the modifications performed, particularly in the control unit for the EXE stage, it is quite impossible to produce an accurate comparison for the effects of the One-Hot Encoding technique. The effects of using slack borrowing can not be easily measured since the contribution of the change in the timing of the blocks on the effect in the speed of the processor cannot be easily separated from the effects of the other techniques.

4.1. Coding Style Changes

With multiplexers being one of the most numerous components, the group explored several ways in implementing it. This included using conditional signal assignment, buffers, and purely combinational logic blocks. Each buffer needed an enable for its input, thereby resulting to an area almost two times that of the other two implementations. AmbitBuildgates recognized the conditional signal assignment (with-select statement) as a multiplexer, and transformed it as such. Similarly, the combinational implementation was transformed into a multiplexer, resulting to the same area. Clearly, the combinational implementation would be more complex than the with-select (especially for numerous signals), therefore, conditional signal assignment was chosen. Shaded rows in tables represent blocks and techniques chosen in the final design of the ARM7.

The group also looked into the effects of different coding styles on the area and the resulting number of glitches for a certain test sequence. The condition checker block was implemented using four different coding approaches, namely: the original which made use of nested case and if-else statements, a single if-else statements, combinational logic, and using conditional signal assignment. **Table 8** summarizes the results.

The if-else implementation and the conditional signal assignment produced the same hardware implementation in AmbitBuildgates. Such an implementation produced a fairly small area and minimum delay even with a considerable number of glitches. The group then elected to make use of the if-else coding style for the condition checker and other larger blocks, such as the EXE control units.

Table 7
Area Comparison Between Multiplexers and Buffers

Component	Area (μm^2)
Buffer (32bits, 4input)	5,160.96
Conditional Signal Assignment (32bits, 4input)	3,070.08
Buffer (4bits, 4input)	645.12
Conditional Signal Assignment (4bits, 4input)	414.72
Combinational Mux (1bit, 2input)	46.08
Buffer (1bit, 2input)	80.64
Conditional Signal Assignment (1bit, 2input)	46.08

Table 8
Effects of Different Coding Styles on Area, Delay, and Number of Glitches

Code Type	Area (μm^2)	Maximum Delay (ns)	Glitches
Nested Case - If-else	662.40	2.9053	4
If-else	685.44	1.3894	15
Combinational	714.24	1.5504	8
Conditional Signal Assignment	685.44	1.3865	15

4.2 Area and Delay of ARM7 Components

Table 9 shows data comparing area for components in both the original and optimized ARM7, with the optimizations as described in the design section. **Table 10** on the other hand, shows data comparing observed maximum delay for the original and optimized components. Delay comparison per block was not done since most of the blocks are changed in such ways (for example, the removal of the BDT Block in the control unit) that accurate comparison with the original is not possible. Note that these figures were taken after the synthesis stage.

Table 9
Area Comparison Between Original and Optimized ARM7 Components

Component	Area (μm^2)
Original Shifter	79,943.04
Optimized Shifter	41,322.24
Carry-Select Adder	9,492.48
Brent-Kung	6,410.88
Kogge-Stone	13,564.80
Original Register File	258,122.88
Optimized Register File	171,118.08
Original ALU Block	18,950.40
Optimized ALU Block with KS Adder	20,839.68
Original Multiply Block	11,508.48
Optimized Multiply Block	10,304.64
Original BDT Offset	15,454.08
Optimized BDT Offset	3,048.28

Table 10
Delay Comparison between Original and Optimized ARM7 Components

Component	Observed Maximum Delay (ns)
Original Shifter	7.42
Optimized Shifter	4.93
Carry-Select Adder	8.94
Brent-Kung	4.37
Kogge-Stone	3.76
Original BDT Offset	19.77
Optimized BDT Offset	7.49
Original Register File (write/read)	1.541 / 2.101
Optimized Register File (write/read)	1.127 / 2.089

4.3. ARM7 Power and Delay Results

It was discovered that data processing instructions proved to be one of the major bottlenecks in the processor's datapath. This is because this type of instruction makes extensive use of almost all the components in the datapath, incurring a large amount of delay. Therefore, two versions of the ARM7 were created: one using a Brent-Kung Adder and another using a Kogge-Stone Adder. The latter proved to be faster, but has greater area due to wiring overhead. Power was analyzed for both, each using two cases: one for their respective maximum clock frequencies and at 100MHz to enable comparison with the original ARM.

Table 11
Power (in mW) of Two Optimized ARM7 Processors per Instruction Mix

Instruction Mix	**ARM7 with BK adder (26ns)**	**ARM7 with BK adder (100ns)**	**ARM7 with KS adder (25.5ns)**	**ARM7 with KS adder at (100ns)**
GCF 245, 252	35.82	9.973	42.63	10.52
GCF 110, 111	34.84	9.896	43.58	10.60
Cuberoot 216	32.94	9.294	33.02	8.766
Cuberoot 1	31.19	8.705	24.76	7.854
Strcmp neg	33.97	8.948	30.12	7.955
Strcmp neg1b	33.50	9.216	25.72	7.969
Blockcopy 1	28.44	7.690	24.21	6.861
Blockcopy 3	25.68	6.822	23.17	4.600
Factorial 12	43.38	11.40	43.17	11.09
Factorial 1	33.29	9.235	25.31	8.189

Table 11 shows the power consumption of the optimized ARM7 for all the instruction mixes tested. The mixes were chosen in such a way that their results would substantially vary from each other, enabling the group to acquire the observed maximum and minimum power consumption. Also, since the PrimePower tool measures average power, the test values in the mixes were chosen to require short and long simulation times.

The mixes which yielded the most power were Factorial 12 and the GCF of 110 and 111, while the least was given by Blockcopy3. The two mixes are dominated by data processing instructions, thus making extensive use of the shifter and ALU. Also, the two mixes repeatedly use a small number of registers, thus consuming large amounts of switching power. It is interesting to note that for the GCF whose majority consisted of data processing instructions, the measured power was larger for the ARM7-KS combination.

Table 12
Power Consumption (in mW) of Different ARM7 Component Blocks

Component	Original ARM7	ARM7 with BK	ARM7 with KS
ARM7	6.115	11.40	11.09
Register File	1.262	2.604	2.411
Shifter	0.2539	2.215	2.110
IF	0.2398	0.4374	0.4335
ID	0.1499	0.6496	0.6190
EXE control	1.626	0.7265	0.6326
ALU Block	0.7925	1.378	1.881
Adder	0.1629	0.4079	0.5922
PSR Block	0.3914	0.5107	0.4155
Multiply	0.2181	0.4618	0.4819

Table 12 gives the power consumption of the major components of the three ARM7 versions. It is important to note that data for this table are taken for the instruction mix Factorial 12 which yields the maximum power. Also, the optimized versions of ARM7 were taken at a frequency of 10MHz. This is in order to establish a comparison between the original and the optimized, in accordance with the equation for dynamic power. The "others" consists of the remaining components of the processor: multiplexers (ALU, BS, Sval, Exception, Bbus and Addr), latches (Incrementer, Datain, Dataout, Base, BS, Address and Carry) and other blocks (Incrementer, Load-Byte and Store-Byte). Also, it is beneficial to mention that the incrementer used in this case is also a Kogge-Stone adder without the carryin.

The bulk of the original ARM7's power was consumed by the EXE_control block, followed by the register file then the ALU. The top power consumers of the optimized version however, are the register file, shifter and the ALU. The EXE_control block now only consumes a mere 6% of the processor's power.

Table 13 now compares power consumption per component at the observed maximum clock frequency of the processor. Notice that the distribution is not much different from the power measured at 10MHz. The register file still consumes the most power, followed by the shifter and the ALU. Also note that the other blocks also consume a significant part of the total power. It is quite surprising though, that the power consumption of the clock generator (with its switching probability being 100%) is not significant. This can be attributed to the fact that the clock generator only has a 1-bit change, as opposed to majority of the blocks possibly having 32.

Table 13
Power Consumption (in mW) Between ARM7 with a Brent-Kung Adder and ARM7 with a Kogge-Stone Adder

Component	ARM7 with BK (26ns), minimum	ARM7 with KS (25.5), minimum	ARM7 with BK (26ns), maximum	ARM7 with KS (25.5), maximum
ARM7	25.68	23.17	43.38	43.17
Register File	4.682	4.436	9.880	9.818
Shifter	3.723	3.269	8.256	8.185
IF	0.7396	0.6524	1.671	1.605
ID	1.197	0.9737	2.453	2.215
EXE_control	2.021	1.714	2.784	2.380
ALU_Block	1.927	2.236	5.222	7.597
PSR_Block	1.619	1.452	1.884	1.508
Multiply	1.661	1.657	1.753	1.764
Clock Generator	0.6603	0.722	0.6604	0.7223
Others	7.4501	6.0579	8.8166	7.3757

Data for the table above was taken for the instruction mixes which give the minimum and maximum power for both cases. Minimum power is given by the Blockcopy 3 instruction, while maximum is given by Factorial 12.

Table 14
Comparison of Performance Metrics

	Area (mm^2)	Maximum Clock Frequency	Observed Minimum Power	Observed Maximum Power
Original ARM7 (no clock generator)	2.0826	10 MHz (100ns)	2.197mW	6.115mW
Optimized ARM7 (no clock generator)	0.98059	45.5MHz (22ns)	24.88mW	44.44mW
Optimized ARM7	0.99570	40 MHz (25.5ns)	23.17mW	43.17mW

Table 14 shows data from the original, the optimized ARM7 without the clock generator, and the final optimized ARM7 with a clock generator. The maximum clock frequency for the version without the clock generator was measured as to provide a fair comparison with the original. It was found that this optimized version of the ARM increased maximum clock frequency by more than fourfold.

The version with no clock generator (hence gets ideal clocks) had a higher clock frequency than the final version because producing the two clocks proved to be another bottleneck in the design process. Using too many buffers produced a very short phase 1, while using too little produced a very small non-overlap time.

Increase in speed for the final ARM7 proved to be very substantial. Power however, also increased by a large amount. With the extensive use of combinational logic and multiplexers, switching for majority of the blocks was frequent. This resulted to dynamic power comprising almost 100% of ARM7's total power consumption.

5. CONCLUSIONS

The optimizations utilized in order to increase speed in the original ARM7 are the following: change in coding style, use of one-hot encoding, use of slack borrowing and architectural modifications. These optimizations resulted to a much faster and hardware oriented implementation. Speed increase was fourfold and area decrease to about half the original. This, however, came at the expense of a substantial increase in power, which increased by around 7 to 10 times.

For further study, the group recommends the following:
1. Testing with a significant number of benchmarks to provide a more stable basis for comparison;
2. The addition of a coprocessor to complete the whole ARM7, since the base is now optimized for speed;
3. Performance evaluation of the base processor with the inclusion of instruction and data caches; and
4. The application of different power-reduction techniques, since the power consumption of the resulting base ARM is considerable.

6. REFERENCES

1. ARM Limited. *ARM7 Datasheet.* Document No. ARM DDI-0020C. http://www.arm.com/techdocs/56QGGJ/$File/ARM7vC.pdf, 1994.
2. S. Furber. *ARM System Architecture.* New York: Addison-Wesley Longman., 1996.
3. J. Delarmente, et.al. *High-Level Implementation of the ARM7 Microprocessor.* Department of Electrical and Electronics Engineering, University of the Philippines, Diliman, March 2004.
4. R. Airiau. *Circuit Synthesis with VHDL.* The Netherlands, Kluwer Academic Publishers, 1994.
5. C. Hamacher, Z. Vranesic, and S. Zaky. *Computer Organization, 5th edition,* New York, NY: McGraw-Hill, 2002.

7. ACKNOWLEDGEMENTS

The group wishes to express their gratitude to Intel Manufacturing, Philippines, the Philippine Council for Advanced Science and Technology Research and Development (PCASTRD-DOST), and the Department of Science and Technology (DOST) for providing the resources needed in completing this project.