A High Performance 32-bit ALU for Programmable Logic

Paul Metzgen
Altera European Technology Center
Holmers Farm Way
High Wycombe
HP12 4XF, UK

pmetzgen@altera.com

ABSTRACT

The Arithmetic-Logic-Unit (ALU) is at the heart of a modern microprocessor, and its size and speed are often significant contributors to the overall processor's cost and performance. This paper presents the design of the ALU used in Altera's NIOS 2.0 soft processor implemented on Altera's Apex 20KE FPGA architecture. This ALU enabled the 32-bit NIOS 2.0 to consume only 1200 LEs and run at 85MHz. This is a 50% size reduction and 70% speed improvement over its predecessor, NIOS 1.1.

The Logic-element (LE) is the basic building block within the Apex architecture. Making full use of the advanced features of the LE has resulted in this novel ALU design. A functional representation of the logic is used to describe how the ALU performs the core set of NIOS instructions, and an LE representation shows the amount of logic-resources needed for the implementation. The cost of additional features such as a barrel-shifter and custom instructions is also described.

Likely worst-case delays for different routing and logic elements are used to estimate the ALU's speed. Further speed and size optimizations are also presented from which it is possible to create ALU ranging in speed from 87 MHz to over 100 MHz.

Categories and Subject Descriptors

C.1.1 [**Single Data Stream Architectures**]: RISC/CISC, VLIW architectures.
C.1.3 [**Other Architecture Styles**]: Adaptable architectures, Pipeline processors.
C.4 [**Performance of Systems**]: Design studies, Modeling techniques.

General Terms

Performance, Design.

Keywords

Programmable Logic, FPGA, soft processors, Nios, ALU, Apex 20KE.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'04, February 22–24, 2004, Monterey, California, USA
Copyright 2004 ACM 1-58113-829-6/04/0002…$5.00.

1. INTRODUCTION

The Arithmetic-Logic-Unit (ALU) is at the heart of a modern microprocessor, and its size and speed are often significant contributors to the overall processor's cost and performance.

This paper presents an ALU that has been designed to take advantage of Altera's Apex 20KE FPGA architecture[2]. This ALU is used in Altera's NIOS 2.0 soft processor[1]. There are two variants of NIOS, one with a 32-bit data path and the other with only a 16-bit data path. This paper will mainly focus on the 32-bit variant as a 16-bit ALU can be constructed using the lower 16-bits of a 32-bit ALU (with some minor adjustments to the barrel-shifter).

The NIOS processor has a 16-bit instruction set, with over 64 instructions operating on a set of 32 registers (in a sliding window of 256 or 512 registers). There are two classes of instructions that use the ALU; some require one operand (RA), whilst others require two (RA, RB). Two operand instructions often have two variants allowing the possibility of using an immediate in place of RB.

The speed of NIOS 1.1 was limited to 50MHz by the ALU, and the ALU also contributed significantly to the overall size of 2400 Logic Elements (LEs). The ALU design presented in this paper enabled the entire NIOS 2.0 processor to be half the size (1200 LEs) and operate at a higher speed than NIOS 1.1 (85 MHz).

Control logic and data path logic are more closely coupled in the design of NIOS 1.1, making it difficult to do a like-for-like LE comparison. The ALU data path presented in this paper, including some additional forwarding multiplexers, accounts for 384 of the 1200 LEs (33%), further logic is also needed for flags, program-control, program counters, address generation, memory interfacing, and instruction-decode which are beyond the scope of this paper. The speed of the ALU was the most significant factor in determining the overall speed of the NIOS processor, however other aspects of the processor also required careful design.

Section 2.1 and 2.2 will describe how an ALU operates within a processor pipeline. The Apex 20KE architecture is described in section 2.3, and section 2.4 describes how to estimate the clock speed of a circuit implemented in this architecture.

Section 3 describes the new high performance ALU that is designed specifically to take advantage of the Apex 20KE architecture. Just like its predecessor, the new ALU is able to implement all of the core set of NIOS instructions in one clock cycle (as shown in section 3.1). Careful attention to multiplexing

with the Apex 20KE architecture details in mind lead to the small and efficient implementation described in section 3.2. Sections 3.3 and 3.4 analyze the critical speed paths within the 32-bit and 16-bit ALUs, whilst sections 3.5 and 3.6 show how it is possible to integrate a full 32-bit barrel-shifter and custom instruction support. Using the further size optimization presented in section 3.7 leads to a fully featured 32-bit ALU implementation that is only 320 LEs in size and can operate at 91 MHz.

It should be possible to operate the 32-bit ALU at speeds over 100MHz with the enhancements described in the further work section 4.

2. BACKGROUND

2.1 ALU Operation

Although the ALU described in this paper is used in NIOS 2.0, it can be targeted for any processor with at least 3 pipeline stages: operand-fetch, execute and operand-write-back as shown in Figure 1.

The operand fetch stage is responsible for retrieving operands from the registers (or immediate) specified in the instruction, and delivering these values to the execute stage. Most instructions require two operands and those values are held in flip-flops RA and RB as shown in Figure 1.

During the execute stage the ALU computes a result based on the opcode of the instruction and the two operands, OperandA and OperandB (a.k.a. OpA and OpB). ‘Forwarding’ ensures that the operand values are up-to-date. This requires two forwarding multiplexers to derive the ALU operands, OpA and OpB, from RA and RB respectively. The result of the ALU is written to the AluResult flip-flop at the end of the execute stage.

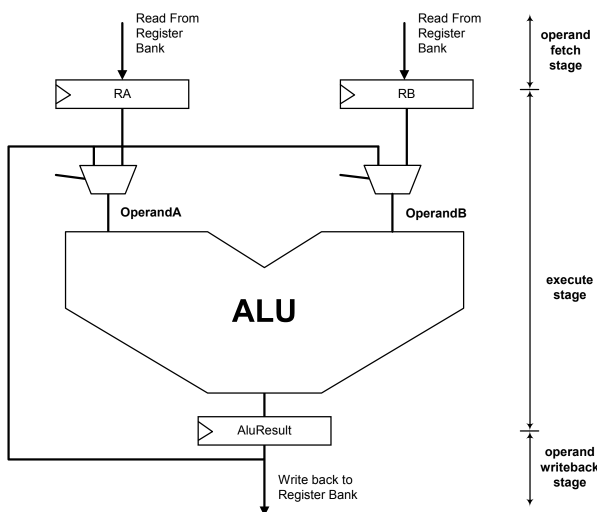


Figure 1: The ALU in a processor pipeline

The operand write-back stage is responsible for updating the correct destination register with AluResult, although not all ALU operations need to write back AluResult to a register (e.g. Compare instructions).

With three stages of pipelining, register banks are actually updated with two cycles of latency. This means that an instruction that requires the result of the previous instruction will read an out-of-date value from the register bank. Forwarding multiplexers are used to update register values that are already in the pipeline.

Forwarding multiplexers ensure that the correct value of a register is used by the time the instruction reaches the execute stage. They can sometimes be omitted to reduce the size or increase the performance of a processor, but care must be taken to avoid ‘data-hazards’. Data hazards occur when an instruction requires the result of a preceding instruction but it is not possible to deliver this result to the execute stage in time. Data hazards can be avoided by identifying instruction sequences that would cause a hazard and then breaking up those instruction sequences by inserting extra no-op (NOP) instructions. Careful analysis is needed to balance the improvement in speed or size from removing forwarding multiplexers with the additional processor cycles (and complexity) incurred by inserting NOP instructions[3][4].

2.2 A Generic ALU

Figure 2 shows the design typically used to implement an ALU, and on which the NIOS 1.1 ALU is based. This ALU design is typical of most processors designed for ASICs and is constructed from separate units, one of whose outputs is selected as AluResult. ALUs designed in this way rarely share functionality or logic between different units, and their performance is reliant upon an efficient implementation of the AluResult multiplexer. Multiplexers are not as efficient to implement on FPGAs, and the ALU presented in this paper has been designed to merge the functionality of multiple units so as to reduce the multiplexing required. This has resulted in significant area and speed benefits on an FPGA.

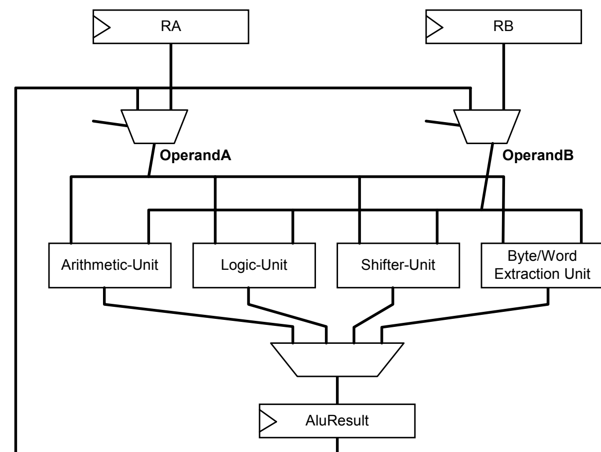


Figure 2: The Nios 1.1 ALU Block Diagram

The ALU is also responsible for maintaining and updating processor flags. Detecting an ALU result of zero (Z-Flag), a carry (C-flag), overflow (V-flag) or negative result (N-flag) are performed by logic in the write-back stage in NIOS 2.0, and are not covered in this paper.

2.3 The Apex 20KE Architecture

NIOS 2.0 was designed for the Altera Apex 20KE architecture[2]. Both the size and the speed of the ALU described in this paper benefit from exploiting advanced features of the Apex 20KE Logic Element (LE).

The Apex 20KE architecture is hierarchical; groups of 10 LEs are packaged into a Logic Array Block (LAB), and 16 LABs (and a RAM block) form a MegaLab.

The Apex 20KE LE can be used in two different modes, 'normal-mode' is shown in Figure 3 and 'arithmetic-mode' is shown in Figure 5. Both modes rely on the two components of an LE, a Look-Up-Table (LUT) and a flip-flop.

In Normal mode, the LUT can be used to implement any function of the four inputs. The LUT can feed the LE output either directly or through the register. The Apex 20KE LE can optionally allow a number of LUTs to be wire-ANDed together using a 'cascade-chain'. The cascade chain is implemented using dedicated logic and routing, so it is both faster and cheaper than a wire-AND implemented in LEs using conventional routing.

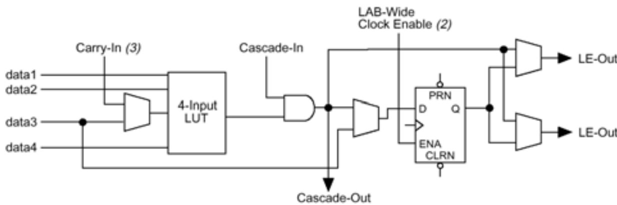
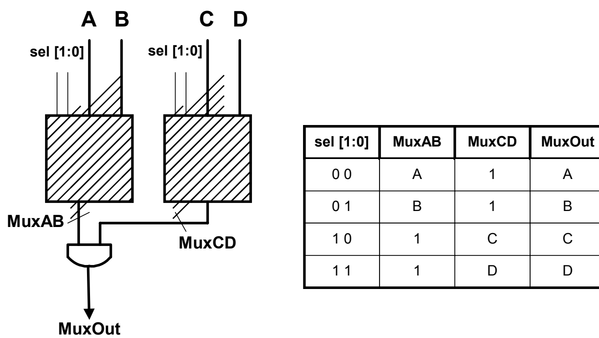


Figure 3: The Apex LE in 'normal'-mode

Figure 4 shows how it is possible to construct a four to one multiplexer in just two LEs using the cascade-AND feature. Both LEs use the binary encoded select lines, and a different pair of the multiplexer data inputs. If the select lines pick A (or B) then MuxAB is set to A (or B) and MuxCD outputs a '1', however if the select lines pick C (or D) then MuxAB outputs a '1' and MuxCD outputs C (or D).



sel [1:0]	MuxAB	MuxCD	MuxOut
0 0	A	1	A
0 1	B	1	B
1 0	1	C	C
1 1	1	D	D

Figure 4: 4:1 multiplexer in 2 Apex LEs

In arithmetic-mode, the LUT is configured to act as two independent functions of 3 inputs. Both are functions of the same three inputs, one of which is the carry-in from the previous LE. Usually, the upper 3-input function computes the arithmetic sum, and the lower 3-input function computes the carry-out from the LE. (The carry-out feeds the carry-in of the next LE in the chain). The sum can feed the output of the LE directly or through the flip-flop.

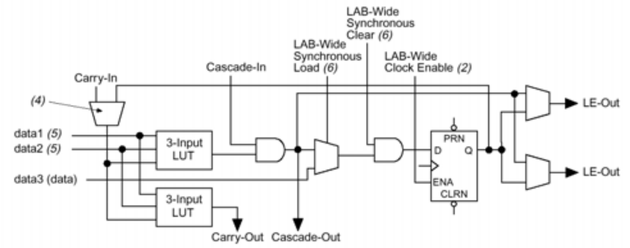


Figure 5: The Apex LE in 'arithmetic'-mode

In both arithmetic and normal modes, the flip-flop (if used) offers additional functionality from a synchronous clear, and a synchronous load. Whilst these 'secondary' signals do not consume LE inputs, they are common to the groups of 10 LEs in a LAB.

As well as arithmetic functions, the carry-chain can be used for other logic functions. Logic functions can be significantly faster when implemented using carry-chains. The ALU presented in this paper will make use of the carry-chain for computing a specialized forwarding function for OpB.

Unlike its successor, Stratix, Apex has no hard multipliers, so NIOS must implement multiplication using a two cycle MSTEP instruction that can be done using the ALU presented, but is not presented in this paper.

2.4 Estimating Fmax

The maximum clock speed of a synchronous design is termed fmax and is calculated as 1000/r2r where r2r is the longest delay of any flip-flop to flip-flop path in ns.

Table 1 shows delays for different routing and LE structures in the fastest Apex device. These delays were obtained from Quartus' timing information on a number of test circuits. For low fanout signals, these numbers were found to be reasonable estimates. Routing in particular can vary so 'likely worst-case' values are used.

Table 1: Apex routing and logic delays

Routing Element	Approx. Longest Delay
Within MegaLab	1.0ns +/- 10%
MegaLab to MegaLab	2.5ns +/- 20%
Logic Element	**Approx Longest Delay**
LUT (input to output)	1.0ns
4:1 Mux	1.3ns
N Cascaded LUTs	(0.7 + 0.3 N) ns
Short Carry Chains	1.5ns
16-bit Adder	3.0ns
32-bit Adder	5.0ns

Table 1 shows that routing is significantly faster if it can be constrained to within a MegaLab (a 160 LE region). Although Quartus' place and route software tries to pack clusters of logic

into the same MegaLab, some clusters can exceed the 160 LE limit, and so are forced to use the more expensive MegaLab to MegaLab routing.

3. AN IMPROVED ALU DESIGN

This section introduces the new ALU design. Section 3.1 uses a functional diagram to explain how the ALU is able to perform the core set of instructions, and section 3.2 shows how this ALU design is implemented efficiently using Apex 20KE LEs.

3.1 Functional Description

Figure 6 presents a functional diagram of the improved ALU.

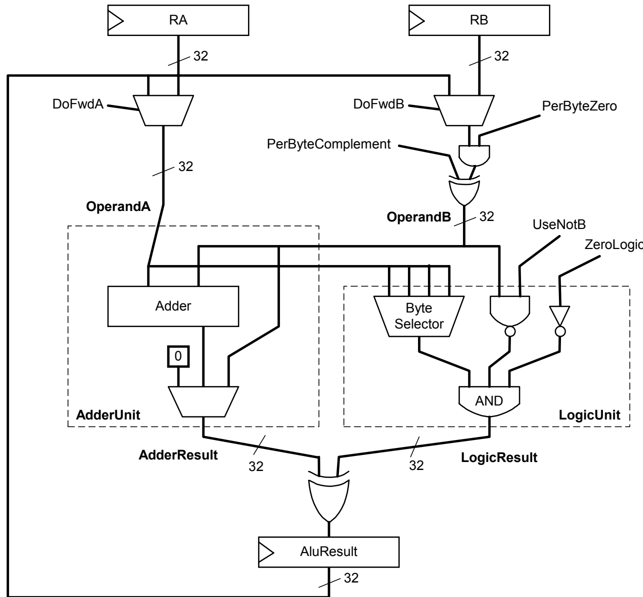


Figure 6: The NIOS 2.0 ALU functional diagram

OperandA and OperandB (a.k.a. OpA and OpB) are the operands to the ALU and are computed from the forwarded values of RA and RB respectively. The PerByteZero and PerByteComplement control signals can be used to manipulate each byte of OpB; each byte can be independently set to RB, not RB (~RB), all zeroes (0) or all ones (-1).

The core of the ALU consists of two functional units, AdderUnit and LogicUnit. Rather than being multiplexed together, the outputs of the AdderUnit and LogicUnit are XORed together. The use of the XOR in this way allows the functional units to work together, rather than independently, and results in a smaller overall ALU. Although the use of an XOR gate in an ALU is not novel [5], section 3.2 will show how the XOR functionality can be combined efficiently with the forwarding logic, and this novel approach leads to improved ALU size and speed in the APEX 20KE architecture.

The AdderUnit can compute OpA + OpB, OpB or 0.

The LogicUnit contains a byte-selector. The byte-selector is actually made from four independently controlled 4:1 multiplexers. Each multiplexer can select any one of the four bytes in OpA. Hence the byte-selector is used to perform rotates and byte/word extractions. The result of the byte-selector can be optionally AND-ed with ~OpB using the 'UseNotB' control signal. A 'ZeroLogic' control signal is used to force the LogicResult to 0.

Table 2 shows how this ALU design can be used to implement most of the common NIOS instructions (whose semantics are described in [1]). Note that the ALU result is the XOR of AdderResult and LogicResult.

For an ADD instruction, the AdderUnit is used to compute the addition, and the LogicUnit is zeroed. XORing AdderResult with zero means that AluResult is AdderResult. Subtracts can be implemented in the same way as ADD but complementing OpB and setting the carry-in of the adder to '1'. Note that the complementation of OperandB is handled by the PerByteComplement control signal. Reading RA into RB and forcing RA to 0 in the operand-fetch stage implements NEG. NEG can then behave just like subtract. The ABS instruction, which calculates the absolute value of a register, is implemented in the same way as a NEG instruction, but AluResult is only marked as valid if the sign of OpB was negative. An 'invalid' AluResult is neither written back to any registers nor used for forwarding.

Logical instructions are implemented using both the AdderUnit and the LogicUnit. For ANDN, AdderResult is set to zero, and the LogicUnit ANDs OpA (from the byte-selector) with ~OpB. The PerByteComplement control signal can be used to complement OpB to implement AND instead.

To perform an OR instruction, the LogicUnit is used to compute (OpA & ~OpB), and AdderResult is set to OpB. It can be shown from a truth table that (OpA & ~OpB) XOR OpB = OpA OR OpB.

An XOR instruction is performed using the XOR-gate; the AdderResult is set to OpB, and the LogicResult is set to OperandA (using the byte-selector). NOT can be achieved by implementing an XOR with OpB set to all ones (-1). (Setting OpB to (-1) can be achieved using both the PerByteZero and the PerByteComplement control signals)

A MOV instruction is implemented in the same way as NOT but with OpB set to all zeroes.

The next few instructions in Table 2 are examples of NIOS' byte and word instructions. Although NIOS has a 32-bit wide data-path, it is able to read and write individual bytes to and from memory. This is achieved with a sequence of instructions. Reading a byte from memory is achieved by reading a 32-bit aligned 32-bit word containing that byte, and then extracting the relevant byte using an EXT8 instruction. For signed bytes, a further sign-extend instruction (SEXT8) is also needed. EXT8 uses the least significant bits of an address in RB to choose a byte to extract from RA as shown in Table 2. In order to write a byte, an EXT8 instruction is used to transfer the byte to be written to the least significant bits in a 32-bit word. The FILL8 instruction is then used to copy this byte across all bytes in a word. Finally a store-byte instruction is used to store the 32-bit word to a 32-bit word aligned address but with only the correct byte-enables asserted. In this way any selected byte can be written to a specified byte-address. NIOS can also read and write 16-bit words to 16-bit word aligned addresses using EXT16, SEXT16, FILL16 and Store-Word instructions.

Table 2: NIOS Instructions

Instruction	OperandA	OperandB	AdderResult	LogicResult
Arithmetic Instructions				
ADD	RA	RB	OpA + OpB	0
SUB/CMP	RA	~RB	OpA + OpB + 1	0
NEG	0	~RB	OpA + OpB + 1	0
ABS	0	~RB	OpA + OpB + 1	0 (result only valid if OpA[31] = '1')
Logical Instructions				
AND	RA	~RB	0	OpA & ~OpB
ANDN	RA	RB	0	OpA & ~OpB
OR	RA	RB	OpB	OpA & ~OpB
XOR	RA	RB	OpB	OpA
NOT	RA	0xFFFFFFFF	OpB	OpA
MOV	RA	RB	0	OpA
Byte/Word Instructions				
EXT8	RA	RB	0	{0, 0, 0, OpA[7:0]} if OpB[1:0] = '00' {0, 0, 0, OpA[15:8]} if OpB[1:0] = '01' {0, 0, 0, OpA[23:16]} if OpB[1:0] = '10' {0, 0, 0, OpA[31:24]} if OpB[1:0] = '11'
EXT16	RA	RB	0	{0, OpA[15:0]} if OpB[1] = '0' {0, OpA[31:16]} if OpB[1] = '1'
FILL8	RA	-	0	{OpA[7:0], OpA[7:0], OpA[7:0], OpA[7:0]}
FILL16	RA	-	0	{OpA[15:0], OpA[15:0]}
SWAP	RA	-	0	{OpA[15:0], OpA[31:16]}
SEXT8	RA	0xFFFFFF00	OpA[7] ? OpB : 0	OpA & ~OpB
SEXT16	RA	0xFFFF0000	OpA[15] ? OpB : 0	OpA & ~OpB

The EXT instructions extract a byte or word from OpA using OpB to select which byte or word is chosen. The EXT instructions are implemented using the byte-selector in the LogicUnit and the ZeroLogic control is used to set the upper bytes to zero. Note that some additional logic not shown in Figure 6 is required to switch the byte-selector controls between OpB[1:0] and the values needed for other instructions.

The FILL instructions are also implemented using the byte-selector, but in this case setting all bytes to be chosen from the least-significant byte or word in OpA.

SEXT8 sign extends the least significant byte in OpA to 32-bits. This is achieved by setting OpB to the mask for the sign extended bits using the PerByteZero and PerByteComplement controls. The LogicResult selects OpA but forces all but the least significant byte to 0. AdderResult is nominally set to the sign extension mask held in OpB; the synchronous clear signal, which forces AdderResult to zero, is used when the sign of the least significant byte in OpA is positive. SEXT16 sign extends the least significant 16-bit word in OpA in the same way. The SEXT instructions require OpA[7] or OpA[15] to control the synchronous clear of the AdderUnit. This can be achieved using an additional multiplexer (not shown in Figure 6).

This section has shown how the core set of NIOS instructions can be performed by the ALU shown in Figure 6. It is interesting to note that the functional units can sometimes be used in different ways to achieve the same functionality. For example, MOV could also be implemented by mirroring the functionality of ADD, but forcing OperandB to zero.

3.2 Efficient Implementation

This section will show how the ALU is implemented efficiently using the Apex 20KE LE.

Figure 7 has retimed the AluResult register back through the XOR gate; so that the outputs of the AdderUnit and the LogicUnit are now registered directly. In addition, the XOR gate for computing the AluResult is replicated on the inputs to both forwarding multiplexers. Replicating the XOR gate in this way is the key to reducing the critical path, enhancing the ALU performance.

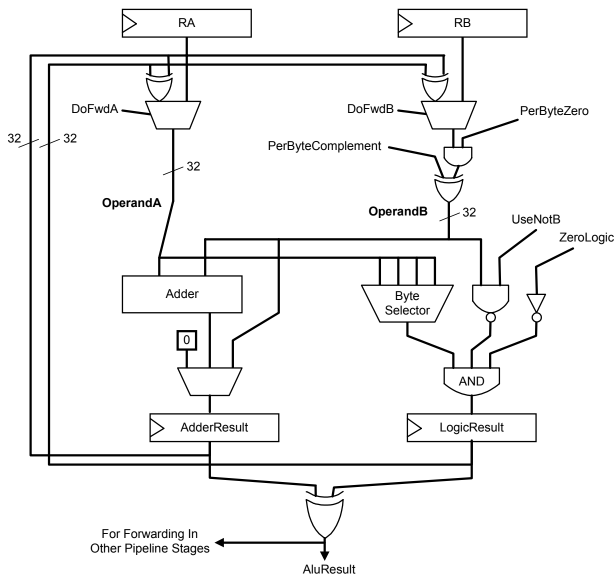


Figure 7: Retiming the AluResult register

The circuit in Figure 7 can be mapped to LEs resulting in the circuit shown in Figure 8. (For simplicity, the shaded boxes are used to represent a single bit-slice of the ALU rather than the full 32-bit data path). Section 2.3 showed that the Apex LE has additional LAB-wide functionality associated with registers; performing this retiming means that the AdderUnit can now exploit this additional functionality leading to a reduced LE count.

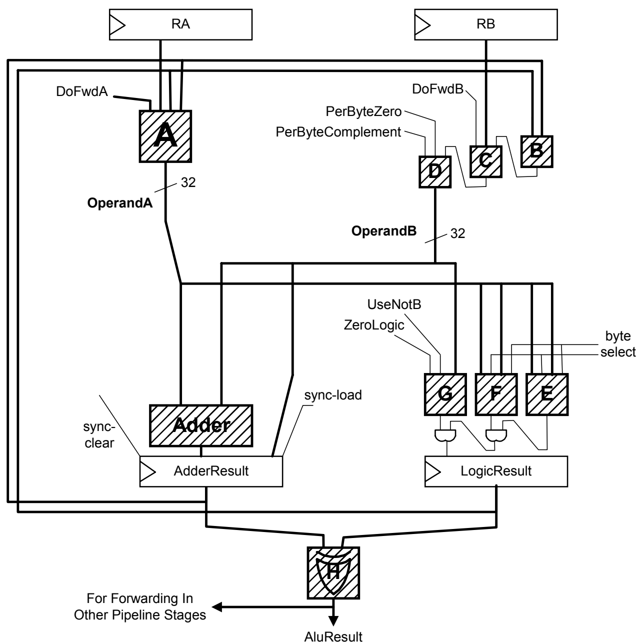


Figure 8: The NIOS 2.0 LE implementation diagram

32 'A' LEs implement all the logic needed to do forwarding on OpA. A carry-chain of 3 LEs ('B', 'C' and 'D') is used to implement the forwarding logic for each bit of OpB. For each bit of OpB, LE 'B's carry-out is the AluResult computed by XOR-ing the correct bit of AdderResult and LogicResult; LE 'C's carry-out is the output of a 2:1 multiplexer used to perform the forwarding on OpB; and LE 'D' is used to implement the PerByteZero and PerByteComplement logic, but this time using the sum-lut so as to feed the output of the LE.

By making use of the synchronous load and synchronous clear signals, the entire AdderUnit can be implemented in just 32 LEs. The synchronous clear forces the result of the adder to zero, whilst the synchronous load is used to bypass the adder with OpB. (This new use of synchronous load on adders is now supported in Quartus-Native Synthesis).

Each bit of the LogicResult is implemented using 3 LEs in a cascade-chain ('E', 'F' and 'G'). LEs 'E' and 'F' implement a 4:1 multiplexer whose output is cascaded with a further LE, 'G'. Note that if 'G' outputs a '0' then the cascade-AND is forced to zero irrespective of the other LEs.

LE 'H' is used to XOR the AdderResult and the LogicResult to give the AluResult that is to be written back to the register file or used for forwarding the AluResult to other pipeline stages. Note that LE 'H' could be implemented using the sum-part of LE 'B' in the OpB forwarding logic. (Every LE in a carry chain has independent sum and carry functions fed by the same three inputs).

The ALU data-path can therefore be implemented using only 8 LEs per bit (256 LEs).

3.3 32-bit ALU Speed Performance

In this section, the speed estimates outlined in Table 1 are used to estimate the speed of the new ALU design.

At least 256 (8 x 32) LEs are needed to implement a 32-bit ALU, but a MegaLab can only contain 160 LEs (16 Labs x 10 LEs/Lab); this means that some routing paths must pay the higher cost for crossing MegaLab boundaries. In this section, it will be assumed that all routing has to cross a MegaLab boundary unless otherwise stated.

Figure 9, Figure 10 and Figure 11 show the three most speed critical paths in the ALU.

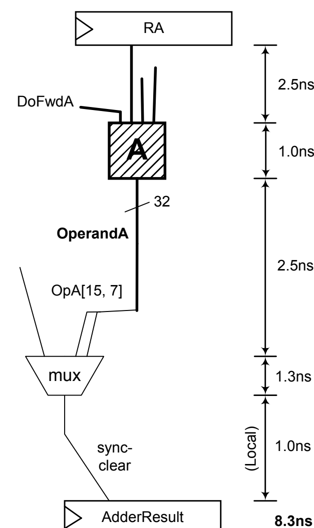


Figure 9: ALU Critical Path

Figure 9 shows the timing from the RA register to the synchronous-clear port of the AdderResult register. A 3:1 multiplexer is needed to implement the Sign extension instructions by clearing AdderResult based on the sign bit in OperandA. This 3:1 multiplexer can be implemented using a cascade-chain. As only one multiplexer is needed it is reasonable to expect it to be packed into the same MegaLab as the Adder, and therefore we can assume that the routing between the multiplexer and the AdderResult register is only 1.0 ns. Overall, this path is 8.3ns that would limit the ALU to 120 MHz.

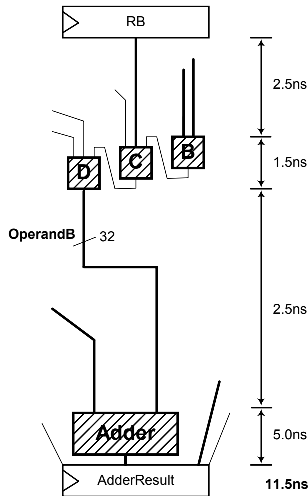


Figure 10: ALU Critical Path

Figure 10 shows the timing from the RB register to the AdderResult through the Adder. The critical path through the adder is from the least significant bit along the carry chain to the most significant bit, making this path a hefty 11.5ns, limiting the ALU's fmax to 87MHz.

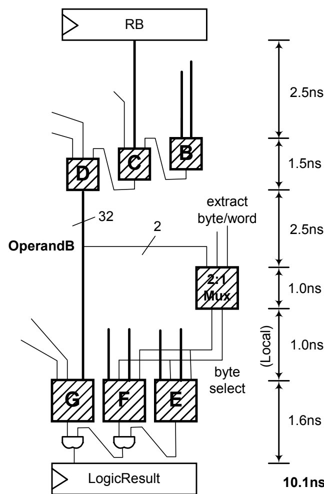


Figure 11: ALU Critical Path

Figure 11 shows the timing from the RB register to the LogicResult register. Note that in order to implement 'dynamic' byte and word extraction, the byte-select control lines need to be controlled from the two least significant bits of OpB. For all other instructions, the byte-select control lines can be set from a flip-flop whose value is based on the instruction opcode (or an immediate). This functionality necessitates an additional 2:1 multiplexer in this path. All four bytes require separate two-bit controls, so only 8 LEs are needed to implement this multiplexing. It can be assumed that this logic would be placed in the same MegaLab as the LogicResult cascade chains, so only a 1.0ns routing hop is needed bringing this path to a delay of 10.1ns (limiting the fmax to 99 MHz).

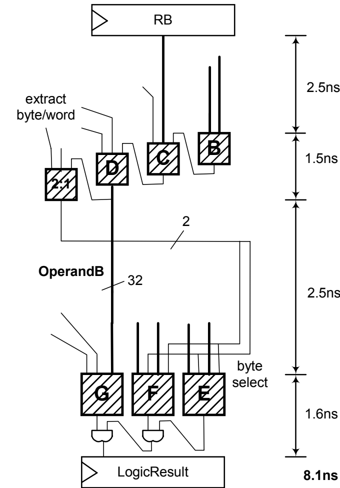


Figure 12: An improved critical path

Figure 12 shows that it is possible to improve the path shown in Figure 11 by implementing the 2:1 multiplexer as an extension to the OpB forwarding multiplexer carry-chain. Only the two least significant bits of OpB need to be extended, however each byte needs to be controlled independently (e.g. for the FILL instructions), so the carry chain actually needs to be extended by a further four LEs (not shown in Figure 12). Each LE implements the 2:1 multiplexer in the sum-part of the LUT, whilst the carry-part of the LUT is used to propagate OpB[1] or OpB[0] unchanged. OpB is now extracted from the middle of the carry chain, which can be accomplished by using the sum part of 'D', as well as the carry-out of 'D' to continue to propagate OpB. This carry-chain trick reduces the delay to 8.1ns (123MHz).

This section has shown how to estimate the speed of the ALU from an approximate timing model. The analysis estimates the speed critical path to be from the RB register to the AdderResult through the Adder, and this path limits the speed of the ALU to 87MHz. In a real place-and-route 90Mhz was achieved, which is close to the estimated value. The difference reflects the fact that inter-MegaLab routing delays typically vary dependant on exact placement and routing.

It is worth noting that apart from paths through the Adder, the next most critical path is 8.3ns. This means that if the Adder delay could be reduced by as much as 3.2ns, the ALU would be able to run at speeds approaching 120MHz. Techniques to reduce the Adder delay are covered in Section 4, but all have implications for the performance of the processor.

3.4 16-bit ALU Speed Performance

The 32-bit ALU contains too many LEs to fit into a single MegaLab, so the worse case routing had to assume that it crossed a MegaLab boundary.

In the 16-bit version of NIOS however, the ALU only costs 128 LEs that will comfortably fit within a MegaLab of 160 LEs. This allows the critical routing to lie within a MegaLab reducing the routing delay from 2.5ns to 1.0ns. From the calculations in Section 3.3, it can be shown that the ALU for NIOS-16 has an overall r2r of 6.5 ns (150 MHz).

At this extreme fmax, other parts of the processor become speed critical. In development, the fastest processor that could be achieved was limited to about 120MHz by the program control unit.

3.5 Incorporating a Barrel Shifter

The NIOS instruction set also supports barrel shifting by both a constant and a register value. In NIOS 1.1, the barrel-shifters were parameterizable so that large shifts were performed over several clock cycles using a small shifter. Typically users configured a shift-by-up-to-7-bit shifter with an early-exit strategy so that shifts of up to 7 took one cycle, shifts up to 14 took two cycles and so on up to 5 cycles for a 31-bit shift. This technique required separate left and right barrel-shifters, and each shifter required at least 3 LEs per bit to implement an 8:1 multiplexer. For Nios 2.0, a full 32-bit barrel shifter is implemented using a total of only 3 LEs per bit, but all shifts take two cycles. The barrel-shifter does not hinder the fmax of the processor, so not much gain could be achieved for allowing the shifter to be configurable.

Figure 13 shows the additional logic required to implement the barrel shifter. OpA holds the 32-bit value to be barrel-shifted left or right, and the least significant bits of OpB hold the shift value.

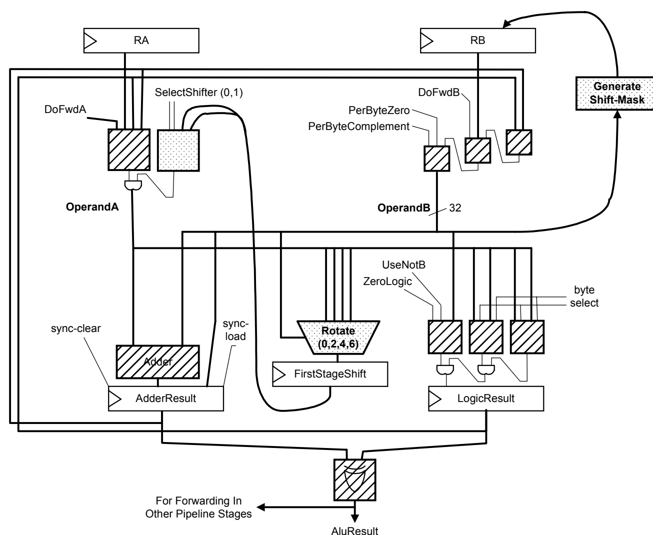


Figure 13: The NIOS 2.0 ALU with Barrel-shifter

A 32-bit barrel shift left or right can be done using a single 32-bit barrel-rotator that rotates to the right. Left shifts by N bits are achieved by rotating right by (32-N) bits. A bitmask is constructed from the shift value on OpB and used to mask off the upper bits of the rotate for a right shift, and the lower bits for a left shift. For signed right-shifts, the bitmask is also used as the sign-extended bits. 32-bit barrel-rotation is done in 3 stages; in the first stage the 32-bit value is rotated by either 0, 2, 4 or 6 bits, the second stage rotates it by either 0 or 1 bits, and the final stage rotates it by the remaining 0, 8, 16 or 24 bits. The final stage can be achieved by reusing the byte-rotator in the LogicUnit; so only the first two stages require additional LEs.

During the first cycle of a barrel shift, the FirstStageShift flip-flop is used to hold the partially shifted OpA. AdderResult and LogicResult are not needed during this stage, as AluResult has not yet been calculated.

During the second cycle of a barrel shift, the partially rotated value held in the FirstStageShift flip-flop must be multiplexed onto OpA, in order to reuse of the byte-rotator. This is achieved without the full cost of using a multiplexer; instead a cascade-chain is used as shown in Figure 13. During the second cycle of the barrel-shift, the OpA forwarding logic is forced to ‘1’. To achieve this, a dummy AluResult of all ones (-1) is computed in the first cycle and ‘forwarded’ during the second. An AluResult of all ones can be achieved by setting the AdderResult to OpB and the LogicResult to ~OpB (as OpB XOR ~OpB = (-1)). When the shifter is not being used, the {0,1} rotator must not interfere with the OpA forwarding logic, so the LE outputs ‘1’.

During the first cycle of a barrel-shift, OpB is used to compute the bitmask. This bitmask will be a sequence of 1’s followed by a sequence of 0’s for a right-shift, or a sequence of 0’s followed by a sequence of 1’s for a left shift. This bitmask is asserted on RB for the second cycle. During the second cycle, OperandB holds the bitmask that is AND-ed with the fully rotated value in the LogicUnit. For signed right-shifts, the bitmask is also assigned to the AdderUnit to be used as a sign extension in the same way as the SEXT instructions. If the sign was positive (or the shift was unsigned) the AdderUnit is forced to 0.

3.6 Custom Instructions

The NIOS instruction set support up to 5 custom instructions. Support for Custom instructions is a key advantage for soft-processors allowing applications to accelerate speed-critical software loops with custom-made hardware. However, it is important to allow custom instruction integration without compromising the fmax of the processor.

Although more advanced schemes exist, a NIOS Custom Instruction takes OpA and OpB and computes a single 32-bit result. Consequently custom instructions have direct access to OpA and OpB.

NIOS selects the output of the custom instruction being executed using a multiplexer. The result of this multiplexer is assigned to the LogicResult flip-flop using the spare input on LE ‘G’ and the synchronous load feature of the flip-flop as shown in Figure 14.

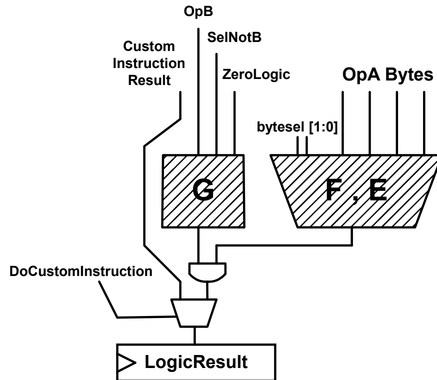


Figure 14: Reading the results into the ALU from custom instructions

In order to maintain the processor speed, it is necessary to design custom instructions that meet or exceed that speed. When this is not possible, as is often the case with complex custom instructions, the custom instruction must be pipelined. To execute at full speed, most custom instructions take at least two clock cycles during which time the entire NIOS 2.0 processor stalls until the result of the custom instruction is available.

3.7 OpB Forwarding

NIOS' instruction set is defined such that most instructions overwrite the register specified by RA. Therefore it is common for code to perform many computations on a single 'scratch' register before writing it back to memory. This means that ALU results are typically forwarded to OpA and seldom to OpB.

By inspection of the code that was generated by the NIOS C-compiler, it was noted that it was very rare for forwarding on OpB to occur. With this observation in mind, the forwarding multiplexer on OpB can be removed in order to reduce the size of the ALU further. Removing a forwarding-multiplexer can create data-hazards, so the processor pipeline ensures that any sequence of two instructions that have a data dependency on OpB are separated by a NOP.

The NIOS 2.0 processor is parameterized to allow the forwarding on OpB to be enabled or disabled.

Removing the OpB forwarding multiplexer reduces the forwarding logic from a three LE carry chain to a single LE (per data-path bit), however, the XOR-gate needed to compute the AluResult can no longer be performed as part of the carry chain, so the overall reduction is only 1 LE per bit.

This brings the overall size of the ALU, including a full barrel shifter, to 320 LEs, and also results in a critical path improvement from 11.5ns to 11.0ns (87 MHz to 91 MHz).

4. FURTHER WORK

The critical path in the ALU is through the 32-bit adder, which represents almost half of the total r2r delay. The next most critical path is 3.2ns shorter, so the adder could be 3.2ns faster before it no longer becomes critical.

It is possible to split the 32-bit adder into an upper and a lower 16-bit adder, reducing the critical path to 9.0ns (111 MHz), but this requires breaking the carry-chain leading to incomplete additions. This section explores three methods that can be used to correct the addition.

Scheme #1: Every ADD (or SUB) instruction is made to take two cycles. If ADDs account for 5% of all instructions executed, this scheme would incur an additional 5% runtime. In the first cycle the AdderResult holds the results of the top-16 and bottom-16 adds. In the second cycle, the 'incomplete' AluResult is forwarded back into the ALU that performs an Add again, but this time OpB is zeroed and the carry from the lower 16-bit adder is passed into the carry-in of the upper 16-bit adder.

Scheme #2: Alternatively, fixing up the AddResult could be performed in the next pipeline stage. Before the upper 16-bits of the AdderResult are XOR-ed with the LogicResult, they are added to any carry-out from the lower 16-bits. (Note that it is possible to implement the XOR-function in the sum part of the adder LEs). Because of the extra adder, forwarding the AluResult directly would reduce the ALU fmax. Instead, the processor must insert NOPs when an instruction tries to use the result of a preceding ADD. However, in all other cases, ADDs take one clock cycle. By inspection, most instructions that follow an ADD instruction do use the result of the ADD, so this scheme is not likely to give an improvement over scheme #1.

Scheme #3: When the carry from the lower 16-bits is '0', then the AdderResult will be correct. This means that the processor only needs to stall and correct the AdderResult when the carry-bit is '1'. Subtracts often result in a carry-bit of '1', so this method should be generalized to perform carry-bit speculation, where the processor guesses the status of the carry-bit between the top and bottom 16-bits, and stalls and corrects upon mis-speculations. Note that for loop counters and array pointers that are unlikely to stray beyond a 65536 limit, carry prediction should be close to 100% accurate.

5. CONCLUSIONS

This paper has presented a novel design for a high performance 32-bit ALU which has been successfully implemented in Altera's NIOS 2.0 processor. The small size and speed of the ALU is the result of a novel design that exploits features of the Apex 20KE LE.

It has been shown how the ALU perform arithmetic, logical and byte/word extraction instructions using only 256 LEs worth of data-path logic (32 x 8 LEs/ bit). With an additional 96 LEs (3 LEs per bit), full 32-bit barrel shifting can be performed in the ALU in two clock cycles without affecting the fmax. This paper has also described how custom instructions are integrated into the ALU. By removing the forwarding multiplexer on OpB, the ALU size can be reduced by 32 LEs and code inspection shows that this is unlikely to cause many extra NOPs to be inserted by the NIOS processor.

The speed estimation numbers predict a performance of 87MHz for the 32-bit ALU, which is close to the 90MHz achieved. Whilst the 16-bit ALU runs at 120MHz as it fits in a MegaLab.

In the further work section, three schemes to increase the fmax to 110 MHz for the 32-bit ALU were outlined. These schemes involved splitting the 32-bit adder. The third scheme is the most promising as it speculates the value of the carry from the lower

16-bits to the upper 16-bits and only stalls to correct the AluResult if mis-predicted.

6. ACKNOWLEDGMENTS

The author would like to thank Tim Allen, the creator of the original NIOS processor, and Peter Hutkins for the help in integrating the ALU into the NIOS 2.0 product.

The author would also like to thank everyone else who supported this project, and in particular Mark Dickinson and Steven Perry for their continued encouragement throughout.

7. REFERENCES

[1] Altera Corporation. *Nios Programmer's Reference Manual*. March 2001.

[2] Altera Corporation. *Apex 20K Programmable Logic Device Family Datasheet*. November 1999.

[3] Hennessy, J.L, Patterson, D.A., Goldberg, D. *Computer Architecture: A Quantitative Approach, 3rd Edition*. Kaufmann, May 2002. ISBN: 1558605967

[4] Patterson, D.A., Hennessy, J.L., Indurkhya, N. *Computer Organization and Design: The Hardware/Software Interface, 2nd Edition*. Kaufmann, Aug 1997. ISBN: 1558604286

[5] Fairchild Semiconductor. *74F181: 4-Bit Arithmetic Logic Unit*. Datasheet, April 1988 Revised September 2000.