ReRISC: A Reconfigurable Reduced Instruction Set Computer

Andrew Huang and Edward H. Kim
Department of EECS,
Massachusetts Institute of Technology, Cambridge

Abstract

The ReRISC processor gives users the opportunity to create application specific instructions for enhanced performance while providing the programming convenience of a conventional RISC processor. The core of the ReRISC consists of an array of 38x8 computational elements, each with 8 configuration contexts that are selectable on a cycle by cycle basis. The computational elements default to the MIT Beta ISA upon soft reset, which reduces redundant reconfiguration cycles. In conjunction with a reconfigurable NOR plane, the core can be wired to perform a wide variety of operations, including vector-style packed word operations, multiply-accumulates, random permutations, tag field verification, and bit field packing and unpacking. This last feature makes the ReRISC better suited for the interpretation of nonnative binaries. The datapath of the 1.8 million transistor ReRISC processor was conceived, designed, implemented and verified in this design project.

1.1 Reconfigurable Hardware

Reconfigurable hardware refers to the broad class of computational architectures between full custom ASICs and programmable processors. According to [Dehon 96], most reconfigurable processing architectures can be characterized by a little over a dozen parameters; this set of parameters is referred to as *RP*-space. Two key parameters are summarized in Table 1.

Parameter	Description
w	Datapath width - number of bit elements controlled by one instruction
c	Contexts - number of instructions stored per group of *w* processing elements

Table 1: Key parameters of *RP*-space.

Some examples will help illustrate the meaning of the *RP*-space parameters. A conventional 32-bit microprocessor with an 8K instruction cache can be thought of as an extreme point in *RP*-space, with $w = 32$, $c = 2048$ (8192 bytes / 4 bytes/instruction), while a typical FPGA can be thought of as the other extreme, with $w = 1$, $c = 1$.

By trying to map various architectures into *RP*-space, one begins to see reconfigurable hardware in a new light. An FPGA can be thought of as a microcoded processor with extremely wide, complex instructions, where an instruction is the configuration bitstream. The high resolution of description provided by the instructions means that the programmer can implement complex algorithms using a few specialized, highly parallel operations. By the same token, it takes much longer to fetch these instructions, and it is much more difficult to compile programs into these instructions. On the other hand, a microprocessor can be thought of as an FPGA with relatively few configurations and a terse configuration bitstream. Because of the limited descriptive power of these configurations, each configuration does less and a complex algorithm has to be implemented with a long string of configurations. However, the time required to fetch a configuration and configure the processor is very short, so the overhead is kept at an acceptable level. Also, because the configurations are so simple, it is easy to compile to a microprocessor. Figure 1 illustrates the instruction width versus instructions executed tradeoff.

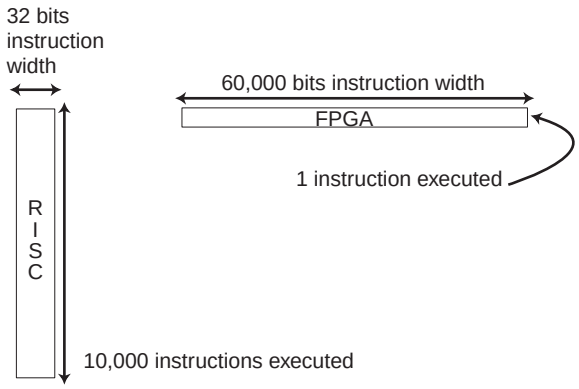

Figure 1: Instruction width versus number of instructions executed for a hypothetical application.

One can see from this model that an FPGA is well suited for applications with a great deal of regularity and parallelism, while microprocessors are well suited for applications with irregular control streams and rapid context switching. The ReRISC, described in section 2, attempts to strike a balance between the two approaches.

2. ReRISC Architecture

As noted in the previous section, compiling to an FPGA is a difficult problem. Also, the size of an FPGA bitstream is very large, so context switching is a very expensive operation in terms of time and memory bandwidth. It would be nice, however, to get the performance offered by FPGAs without the overhead and difficulty of use associated with FPGAs. The ReRISC processor described in this section combines aspects of conventional microprocessors and FPGA technology in an attempt to get the best of both worlds.

2.1 Software Architecture

The ReRISC programming model consists of 32 registers, each 36 bits wide, and a 36-bit program counter. The extra 4 bits per register are provided in case the programmer wishes to use tagged datatypes, hardware semaphores, or instruction-level security. The tag field is 4 bits wide because of the high availability of 36-bit memories. Also, register number 32 is hard-wired to 0x00000000 for programming convenience.

The ReRISC machine state also includes an instruction set configuration (ISC), in addition to the registers and PC. The ISC describes the current instruction set architecture (ISA) of the ReRISC. The ISC consists of 8 rows of 8-context, 38 bit wide instruction definitions and an opcode map. The opcode map defines which opcode activates which set of rows and contexts within the ReRISC, as well as the state of a few internal datapath multiplexers. An opcode can activate anywhere between 0 and 8 rows of computational elements simultaneously in any combination. An active computational element can take on the personality stored in any one of its 8 context cache entries on a cycle-by-cycle basis.

Although the instruction set is configurable, there are still a few hard-wired instructions: those required for fetching data from memory and writing it to the computational array.

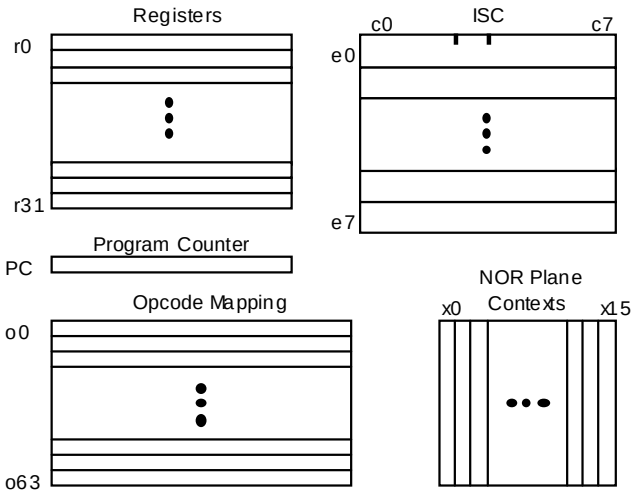

Figure 2: ReRISC machine context summary.

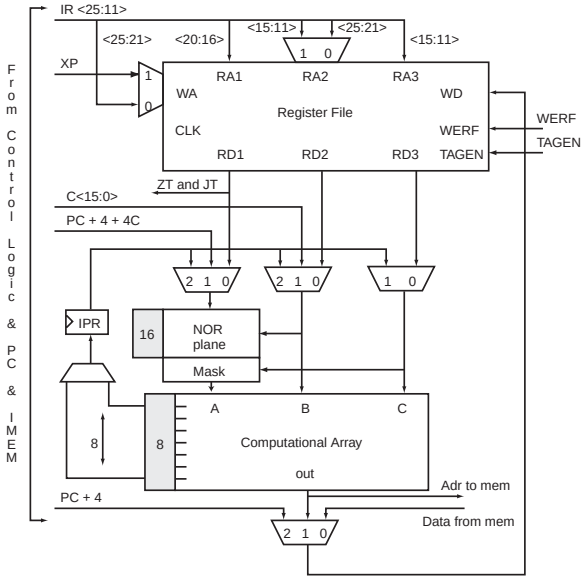

Figure 3: Block diagram of ReRISC datapath.

The ReRISC was designed to run a typical preemptive multitasking operating system. Each process or thread has a machine state associated with it, consisting of the PC, registers, and the ISC. On a context switch, the machine state of the current process is saved and the new machine state loaded in from memory.

To reduce configuration times, a soft-reset of the ISC does not clear all values to zero; instead, it sets the configuration memories to the basic MIT Beta ISA plus the hard-wired instructions required to reconfigure the processor. Thus, most applications will only need to write one or two instruction definitions before running, saving on time during context switches.

Users compiling programs for the ReRISC can take two approaches. The more conventional approach uses a standard compiler with a back-end targetted at a predetermined ISC. The primary advantage of this approach is that the ISC can be tweaked for a particular language, thus yielding more optimal code. In the less conventional but more powerful approach, a compiler analyzes a program to determine the best ISA. It then creates an ISC for that ISA, and then compiles the program into a binary using the optimal ISA. Of course, users can always hand-code an ISC to squeeze out every last drop of performance for a particular application.

2.2 Hardware Architecture

The ReRISC hardware architecture consists of a 32 x 36 3R/1W register file, a 32 x 32 x 16 NOR plane which doubles as a crossbar and barrel shifter, and an 8 x 38 x 8 computational array. Figure 2 provides a block diagram of the ReRISC datapath elements.

1. all of the pointers contained in this section refer to the directory, /afs/athena.mit.edu/user/other/s2001s/Public/6.371/finalproject/, which will be denoted as "~/"

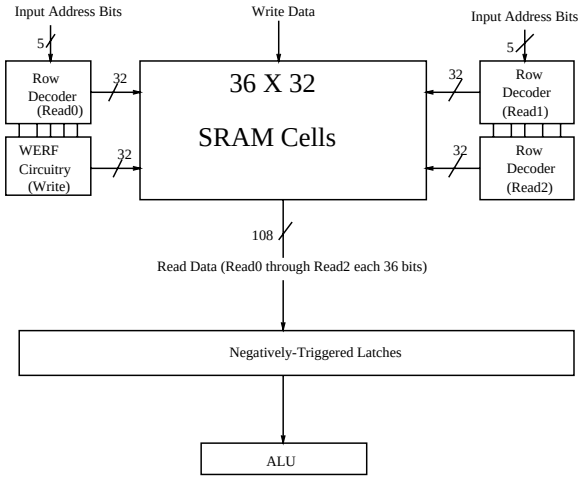

Figure 4: Register file detailed block diagram.

3-READ/1-WRITE Register File (Ed Kim)[1]

Register files are fast RAMs with multiple read and write ports [3]. In this design, the register file consists of 36 registers (R0 through R31), each a 36-bit register, with the last register (R31) being a "zero" register, i.e., only zero read from it and writes are ignored - this register is necessary to execute some of the Beta instructions that require a zero operand or discard results of an operation.

The register file for this version of the Beta has three read ports and one write port, along with a tag-enable port which enables the register file to handle tagged datatypes (the tag-enable port has been omitted for this final project). The READ process involves purely combinational logic, while the WRITE process is clocked. Included with the SRAM cell in the register file are the row decoders, the clocked write-enable signal circuitry, and a latch for simultaneous READ/WRITE operations (the block diagram is in ~/Diagrams/rf.fig). The layout of the register file can be found in ~/Layout/rf.mag.

One-Bit SRAM Cell

The register file uses static RAM cells, instead of dynamic RAM cells, since the access of a register does not guarantee refresh of dynamic cells, thus making it possible for a register to lose its value. The design of the single bit cell follows the standard 6-T SRAM cell design [3], with the following modifications (Schematic of the SRAM cell is in ~/Diagrams/sram.fig, and the layout in ~/Layout/ram.mag):

- An inverter was inserted between the cross-coupled inverters and the two read ports, rd1 and rd2, in order to eliminate any possibility of state change. Without the inverter, the read pass-transistors would share charge with the cellbar node, thus reducing the node voltage and making state change possible.
- A differential write port (two pass-transistors) is used, instead of a singled-ended write port, in order to ensure correct write operations.
- Sizing: all of the inverters are minimum-sized (PMOS: 8/2, NMOS: 3/2), and all of the pass-transistors were enlarged (NMOS, 9/2) in order to mitigate body-effect, which prevents READs from reaching all the way up to Vdd; it reaches 3.7 volts.

Simulations of the SRAM cell in HSPICE and IRSIM verified its functionality and showed a READ time (i.e., rise and fall times involving a READ operation) of approximately 2ns, and a WRITE time of 1 - 1.5ns (Refer to the simulation file, ~/Simulations/ram.tr0). The single-bit SRAM cell was then replicated to form a 36 X 32 array (32 bits plus 4 bits of tag-field bits).

Row Decoder

The ReRISC architecture calls for four separate row decoders, three for READ addresses and one for WRITE addresses. The row decoders take in five address bits (a0 through a4), and generate 32 row select signals (word0 through word31). The file, ~/Diagrams/decoder1.fig, shows a partial schematic of the row decoder. The layout can be found in ~/Layout/decoder1.mag. Two simple yet effective circuit optimizations were added to the design in order to improve performance and minimize area:

1) Speed: Because of the large fan-in associated with the brute-force implementation of the row decoder, a predecoding scheme was used and with it, all of the logic gates of the row decoder have only two inputs.

2) Area: The straightforward approach is to implement the decoders with AND gates, but with the use of DeMorgan's law and negative logic, the number of transistors was drastically reduced.

~/Simulations/decoder.ps contains the IRSIM simulation of the row decoder.

Write-Enable (WERF) Signal Circuitry

The WRITE address decoder differs from the READ decoder in that while the latter involves only combinational logic, the former requires a clocked circuitry. This is because a WRITE operation is permitted to take place only on the rising edge of clock. Contrary to this, a READ operation can happen at any time as long as the read-enable signals (read0 through read2) are asserted. The file, ~/Diagrams/werf.fig, shows a schematic of this WERF signal circuitry: a d-flipflop (dFF) is used to sample WERF at the rising edge of CLK (simulation of dFF is in ~/Simulations/dff.tr0). The output of the dFF and CLK_N ("clock bar") are used as inputs of an AND gate; CLK_N in-

stead of CLK is the input to ensure that the WRITE happens on the falling edge of the clock. The layout of this dFF-CLK_N gate can be found in ~/Layout/WERF.mag. The output of this AND gate, called wp, along with one of the 32 address word from the WRITE decoder, feeds into a second AND gate, called r_sel_AND (layout in ~/Layout/r_sel_AND.mag) whose output is the write-enable signal of each register. Because wp drives only one row select gate at any one time, one wp signal was used instead of 32 separate wp signals.

Simultaneous READ/WRITE operation

In the ReRISC architecture, simultaneous READ/WRITE is one of the required functionalities. By simultaneous READ/WRITE, we mean that when both read-enable and write-enable (WERF) signals of a register are high, and then CLK goes high, the change of content of the register must show up at the falling edge of the next clock cycle, not immediately. In other words, the register file must not commit to the new value of the register until the next clock cycle. Refer to the timing diagram, ~/Diagram/timing.fig. In the diagram, we see Read1 reading out d[A1], the content of register A1. At the same time, the write port is writing "FOO" into the same register. What comes out of Read Data 1 illustrates the desired functionality: d[A1] is read out until the next clock cycle at which point, the new data, FOO, is read. The mechanism called for here is a negatively-triggered latch, which would hold the old value of the register while a new value is being written in. A row of 108 such latches, one for each of the 3 read data signals for 36 cells, is placed between the register file and the ALU.

1/2 PLA NOR Plane (Andrew Huang)[1]

The 1/2 PLA NOR plane is a 32 x 32, 16-context array of NFET pullups in the wired-NOR configuration. The NOR plane is capable of performing a large number of useful operations, including zero testing, random permutations, bit shifts and rotates, and bit packing/unpacking. In order to cut back on the number of contexts required to make the NOR plane useful, the barrel rotate function is also hard-wired into the array. A high priority was placed on the ability to do fast bit-twiddling since one of the more interesting applications of the ReRISC is an emulation processor capable of executing several flavors of nonnative binaries simultaneously.

Common Cell Structure

The NOR plane is an array of cells. Each cell contains a 16 x 1 configuration context memory, a barrel rotate decoder, and some NFETs to do the actual computation. The configuration memory is very similar to that used in the computational array, described later in this report. The barrel rotate decoder is a simple piece of logic which first determines if the NOR plane is being used as a barrel rotator, and if so, what the state of the NFET pulldown should be given a shift code. The decoder supports right and left shifts, as well as vectored shifts (i.e., simultaneously shifting four bytes or two 16-bit words packed into a single 32-bit long word). In order to avoid the poor performance associated with using a ratiod PFET pullup to precharge a heavily loaded bitline, the NOR plane uses a clocked precharge-evaluate style of logic. The bitlines are precharged by a strong PFET pullup on the first half of the clock cycle, and the NOR plane evaluates on the second half of the cycle. Simulations and layouts of the cell structure can be found in ~/rerisc/xbarcell. In general, I have prepared postscript dumps of all the relevant files for your convenience; just look for the files ending in .ps.

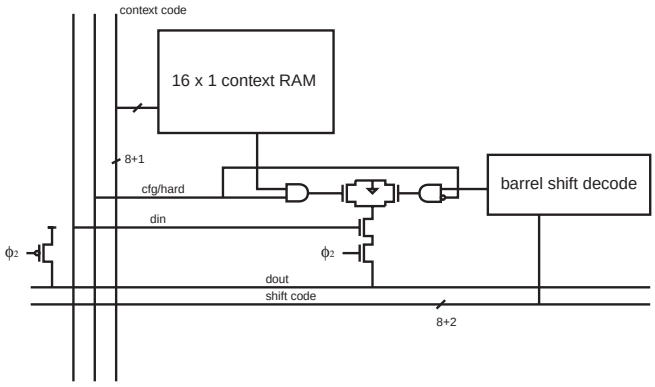

Figure 5: Crossbar cell block diagram.

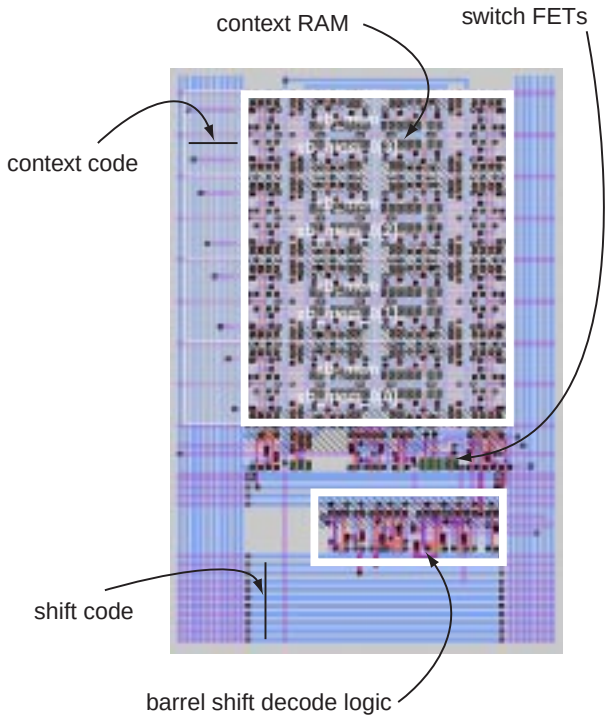

Figure 6: Crossbar cell layout.

1. all of the pointers contained in this section refer to the directory, /mit/bunnie/Public/6.371, which will be denoted as "~/"

Computational Array (Andrew Huang)

The CA is a 38 x 8, 8-context array of computational cells. Each cell features a 4-1 lookup table (LUT) and two 3-1 LUTs for computation. The 4-1 LUT computes the final result of the cell, while the 3-1 LUTs drive the propagate/generate inputs of antiparallel Manchester carry stages. This cell architecture provides sufficient computational power to implement one element of an array multiplier. In addition to the LUTs and carry chains, some ancillary logic is provided to improve the flexibility of the CA. The structure of a computational cell is detailed in figure 7.

The computational array is capable of implementing the entire MIT Beta ISA, sans multiply and divide instructions, with less than 15 single-row contexts, leaving 49 row-contexts for instruction set extensions. Some examples of instructions set extensions that the CA can easily implement include vectored (i.e., four 8-bit ops per cycle) saturating adds, vectored non-power of two precision multiplies, multiply-accumulates, arbitrary bit maskings and tests, and decrement-conditionals. Note that a 32 x 32 multiply requires 4 row-contexts (see section titled In-Place Computation Register) and completes in 4 clock cycles.

LUT structure

The LUTs consist of an 8 bit by 2^n row memory which feeds a 2^n to 1 multiplexer. The n select inputs of the multiplexer are the computational inputs of the LUT. The configuration context is specified on a cycle by cycle basis through the context code. See figure 8 for a block diagram of the LUT.

The memory elements of the LUT are 9T-SRAM cells, consisting of a pair of cross-coupled inverters, a read bitline driver, read and write pass transistors, and a state preset pass transistor (figure 9). The cross-coupled inverters are asymmetrically

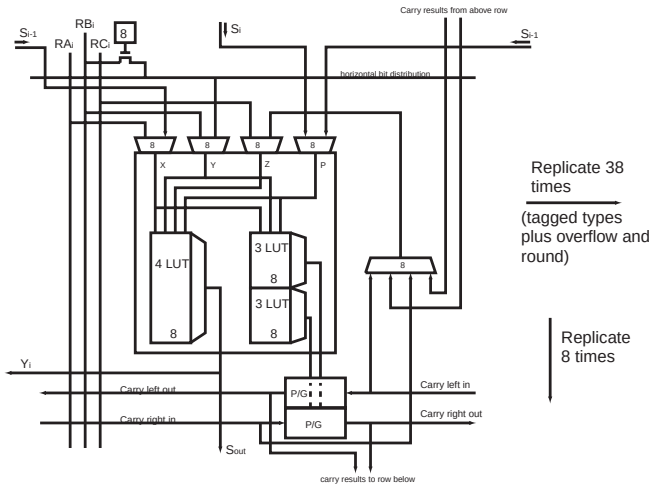

Figure 7: Computational cell block diagram.

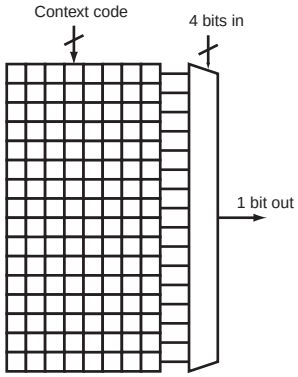

Figure 8: 4-1 LUT diagram showing context RAMs.

sized so as to allow jam-style writing. The read bitline driver is provided to insure that the SRAM state does not change even when driving heavily loaded read bitlines. The decision to use a bitline driver was motivated by the fact that a sense amps are not area-efficient when dealing with memories around 8 or 16-bits in size.

The state preset pass transistor is a FET between the write-side of the cross-coupled inverters and either Vdd or GND (PFETs for Vdd, NFETs for GND). The state preset pass transistors allow the ReRISC to take on a useful default configuration, such as the Beta ISA, after the application of a reset signal. This circuit trick saves users many hundreds of configuration cycles at the cost of one transistor per SRAM cell. The user savings are based on the assumption that a typical application uses a stock set of instructions with only a couple of application specific instructions.

Schematics, layout and simulations (Protel spice and IRSIM) of the SRAM cell can be found in ~/rerisc/cfgmem/.

Configurable Antiparallel Manchester Carry Stage

Two Manchester carry chains are provided per row of computational cells. The chains run in opposing directions, hence the term “antiparallel”. The Manchester carry stage blocks are labelled “P/G” in figure 7. The structure of these carry blocks are detailed in figure 10.

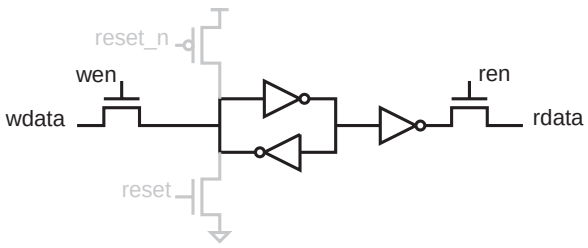

Figure 9: 8T SRAM cell detail.

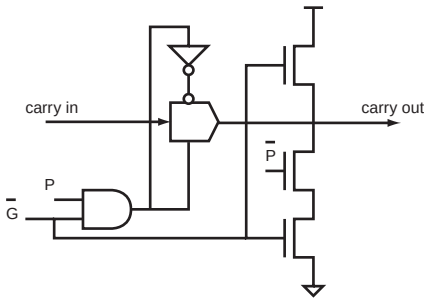

Figure 10: Manchester carry stage detail.

The carry stages provide a means for the fast, bidirectional propagation of intermediates in operations such as addition and wide boolean logic functions. The bidirectional capability is important for the implementation of saturating arithmetic. The carry chain can be split at any arbitrary point to allow multiple reduced precision operations to be performed in a single row of computational cells. This feature is important for implementing vector-style instructions ("packed operations" in Intel MMX terminology). Breaking the carry chain is accomplished by configuring the 3-LUTs to always force propagate inactive, and forcing generate to the correct default polarity, i.e., additions require the zeroth-bit carry in to be 0, while subtractions require the zeroth-bit carry in to be 1. Note that the carry-chain implementation is conflict-free, in the sense that at no time will a pullup be fighting against a pull-down. Also, transistor sizing and the distribution of buffers is designed conservatively because the length of the carry chain is unknown at design time.

Ancillary Logic and Connectivity

A set of multiplexers and pass-transistors are provided to augment the flexibility of the computational cell. Multiplexers with independent 8-context configuration memories allow each cell to be connected to either register file ports, the output of other computational cells, or to various points on nearby carry chains. A pass gate with configuration memory also allows for the connection of a vertically distributed register file bit to a horizontal wire. This pass gate, along with the option to select the output of the upper-right computational cell are important for the implementation of an array multiplier.

In-Place Computation Register

One of the design goals of the ReRISC is the ability to implement the MUL instruction of the MIT Beta ISA. The MUL operation is a 32 x 32 bit multiply with a 32 bit (LSB) result. Thus, the straightforward ReRISC implementation would require a CA which is 38 x 32 in size. For various reasons mostly related to area efficiency, this array size is undesirable. However, if one is willing to make the tradeoff of waiting four cycles to complete a multiply, one can perform the instruction in an array 1/4 the size by taking advantage of the cycle-by-cycle context switching capability of the CA.

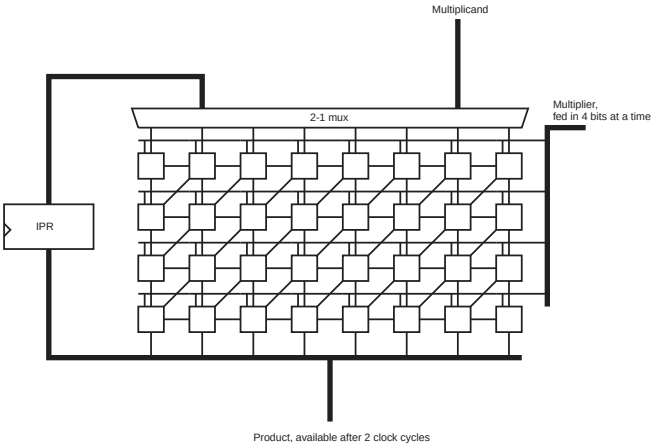

Figure 11: multiplication using the IPR.

This is possible because the array multiply operation falls into a class of computation known in the DSP world as "in-place operations". Certain varieties of the FFT are in-place operations in the sense it can be split temporally into a sequence of operations that use the same piece of hardware over and over. A 32 x 32 multiply can be split into a sequence of four 32 x 8 partial-product summations where the output of the CA is fed back into the CA on successive clock cycles. The configuration of the CA has to change slightly on successive clock cycles (most notably, the horizontal bit distribution pass transistors have to change state), but this happens with no performance penalty. The IPR allows for the feedback of computation results without disturbing the contents of the register file.

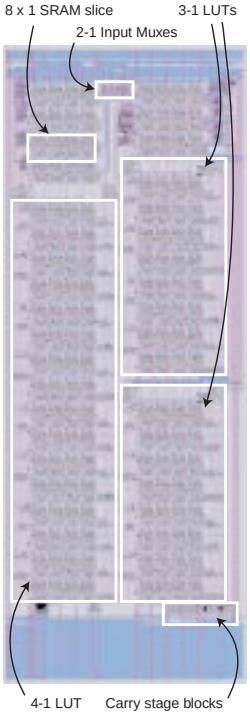

Figure 12: Layout of the computational cell.

3. Verification

All the leaf cells of the ReRISC processor were verified using either a flavor of SPICE or IRSIM. All the leaf cells have been tested for base-level functionality. Pointers to the simulation files for the leaf cells were given in each of the sections of this report that described a major sub-block of the ReRISC.

Some higher-level verification was performed as well. For example, the register file was tested, and so was the crossbar cell and the basic components of the computational cell. However, due to the complexity of the design, it proved to be computationally intractable to verify the entire design within the scope of this class. A rough estimate of the number of transistors in this design is around 1.8 million; at this level of complexity, it takes magic ten minutes to simply run the design rule checker on the design, and a minute or two just to redraw the screen.

4. Conclusion

The datapath of the 1.8 million transistor ReRISC 32-bit microprocessor was conceived, designed, laid out and verified at the leaf node level in this project. The ReRISC was designed around the philosophy of giving users performance where they need it, and convenience when they want it. Its fully configurable instruction set defaults to the simple MIT Beta RISC ISA. Users can then replace or add instructions specific to their application for enhanced performance. The ReRISC architecture is flexible enough to implement a wide variety of specialized instructions, from random permutations to small vector operations, to bit packing and unpacking. The last example, bit packing and unpacking, is especially important for the emulation of nonnative binaries.

One of the major design goals of the ReRISC was to come up with a reconfigurable architecture which has a relatively small reconfiguration overhead. This design goal was met using a number of techniques. First, all the configuration memories in the ReRISC are preprogrammed to default to a useful state upon the application of a soft reset. This means that most users will only have to modify a few contexts specific to their application, instead of having to upload the entire base ISA plus their custom instructions. Second, associated with each computational element are several configurations (multicontext architecture). Each of these cached configurations are available on a cycle-by-cycle basis. As long as one does not exceed the capacity of the context memories, switching between configurations is very fast. The context memory was designed to be large enough such that a typical operating system would only need to modify the ISC on a context-switch, which happens relatively infrequently. In addition to architectural optimizations, some logic/physical design tricks were used to enhance performance, such as the bidirectional, segmentable Manchester carry chain used in the computational array.

One optimization for the ReRISC architecture that could have been applied to save a significant amount of area would be to increase the ratio of bits of computation per configuration memory. Currently, each bit slice in the datapath has its own independent set of configuration memories. I conjecture that

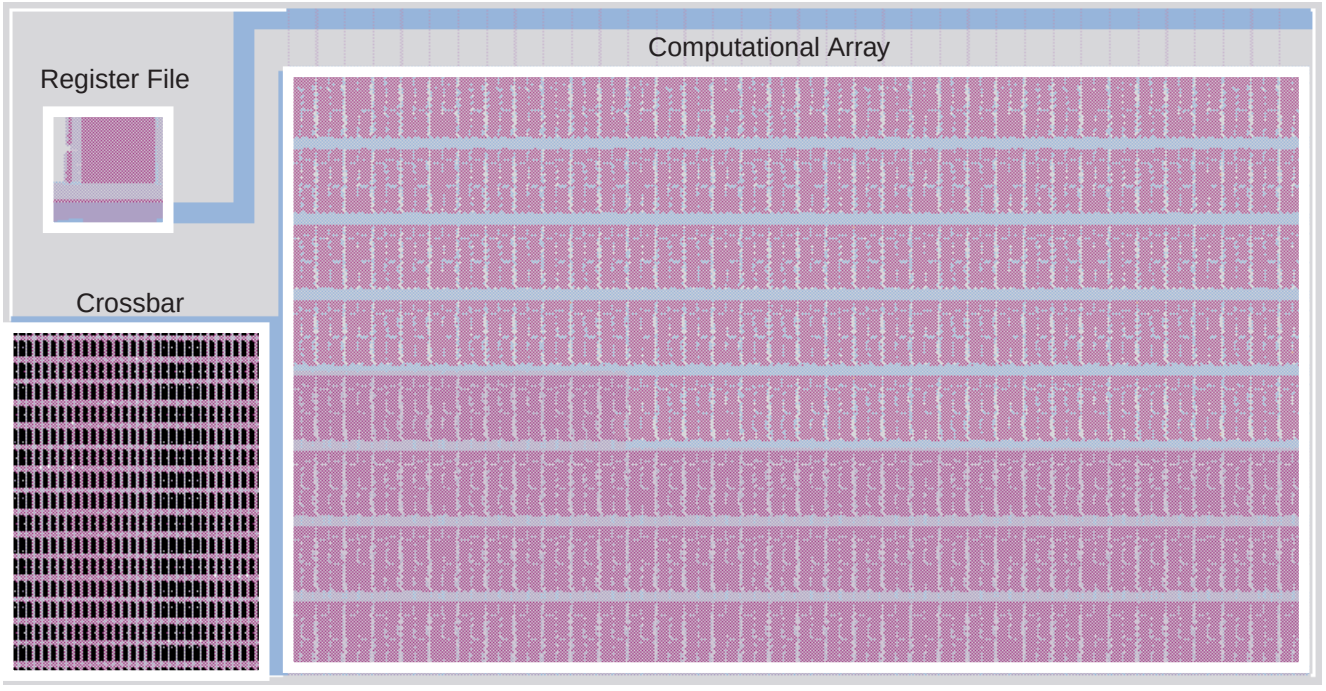

Figure 13: Layout of the ReRISC datapath.

little flexibility would be lost if two bits of datapath were configured by one set of memories. This would cut the area of the memory-dominated computational array down by almost half, and consequently save 30-40% in total die area. A similar processor under development at Berkeley called Garp employs this idea to conserve area as well. [Hauser 97]

On a final note, the ReRISC offers a unique advantage over conventional processors. Contemporary architects are finding themselves having more transistors than they know what to do with, and CPU core frequencies skyrocketing while memory bus speeds continue to lag behind. As a result, designers have been dumping more transistors into larger caches. Cache sizes have been increasing steadily to the point of diminishing returns. The ReRISC is different in the sense that more computational elements could translate directly to higher performance without the complications of superscalar technology. One example is the in-place multiply. Currently, it takes 4 cycles for the multiply to complete. However, if one doubles the size of the computational array, the multiply will complete in 2 cycles. Thus, not only does an architecture like the ReRISC benefit from shorter gate delays as line geometries scale down, it also benefits from the greater density offered by smaller process geometries.

5. References

[1] Jolly, Richard D., “A 9-ns, 1.4-Gigabyte/s, 17-Ported CMOS Register File,” IEEE JSSC, Vol.26, No.10, October 1991.

[2] Shinohara, Hirofumi, et al., “A Flexible Multiport RAM Compiler for Data Path,” IEEE JSSC, Vol.26, No.3, March 1991.

[3] Weste, Neil, Principles of CMOS VLSI Design: A System Perspective, 2nd Edition, Addison-Wesley, 1993.

[Dehon96] Dehon, Andre. “Reconfigurable Architectures for General-Purpose Computing”. A.I. Lab Technical Report No. 1586. October 1996.

[Hauser97] Hauser, John R. “The Garp Architecture”. UC Berkeley Technical Report. October 1997. Obtained from BRASS project website.