6.375 Final Report

Marcus Boorstin and Valerie Sarge

May 11, 2016

1 Project Goals

Our 6.375 final project was to emulate the Apollo Guidance Computer (AGC) on the FPGA. The AGC is a relatively complex processor, so our main goal was to successfully implement the full instruction set and pass a relatively complete verification suite. Such an emulator should easily outperform the original in speed, size, and power draw, so we decided upon 10 million instructions-per-second as a reasonable loose target: this corresponds to a two-order-of-magnitude increase in speed over the original. Additional "stretch" goals included making the processor cycle-accurate and implementing full interrupt and timer functionality in order to run original guidance programs and perhaps even simulate a successful launch.

2 Background

2.1 History

The original Apollo Guidance Computer was conceived as a system to provide guidance, navigation, and control for NASA's Apollo moon missions. At its core was a processor with a clock cycle of approximately 11 µs and a memory bank including 2048 16-bit words of read/write memory and 36,684 words of read only memory (words were composed of one parity bit, one sign bit, and 14 magnitude bits). The AGC used many cutting-edge technologies; it was one of the first to use integrated circuits and pioneered the use of core rope memory for ROM (read/write memory was still implemented as magnetic core). The original weighed 70 pounds and drew 70 watts; an FPGA design can do much better.
In order to emulate an AGC, an FPGA-based system must also provide the surrounding infrastructure, including memory and I/O channels. In addition to the distinction between read-only and erasable memory, implementing the switched-bank architecture is complex due to memory overlaps. Port management is a significant concern as well, with the need to read three registers and an I/O channel to determine the correct bank as well as occasional double reads and writes.

2.2 Assembly Language

There are 49 registers, each of which is distinct and some of which must be dealt with specially. (For example, some registers are incremented when read from, and while most registers

are 15 bits, the `A`, `L`, and `Q` registers have 16 bits for overflow detection and correction). Likewise, the instruction set contains on the order of 50 distinct instructions, depending on how special cases are counted. The basic instructions are laid out below. The argument of the instruction, when present and an address, is referred to as K. Arithmetic is assumed to be 1's complement unless stated otherwise.

Instruction List: Basic Instructions	
TC	Jump after storing a return address in the Q register. Special cases include INHINT, which disables interrupts; RELINT, which resumes interrupts; and EXTEND, which flags the next instruction as an extracode instead of a basic instruction.
TCF	Jump to a location in fixed memory.
CCS	Stores the diminished absolute value of a location in erasable memory into the accumulator, then jumps up to three instructions depending on the original value.
DAS	Double-precision AS; adds the contents of A,L and a pair of locations in erasable memory, then stores the result in those erasable memory locations.
DXCH	Double-precision; exchanges the values in the A,L register pair with those in the K,K+1 address pair.
LXCH	Exchanges the value in the L register with the value at K.
XCH	Exchanges the value of A with the value at K (in erasable memory).
INCR	Increments the value at the given address by 1.
AD	Adds the value at the given memory address into A.
ADS	Computes the sum of A and the value at the erasable memory location K and stores it in both A and K.
MASK	Bitwise ANDs the value at K into the accumulator.
CA	Moves the contents of K to the accumulator.
CS	Moves the 1's complement of the contents of K into the accumulator.
TS	Copies the accumulator to memory and leaves a +1 or -1 in the accumulator if there was positive or negative overflow, respectively; otherwise, it leaves the accumulator unchanged.
INDEX	Causes the value at K to be added to the following instruction before decoding. However, if K is octal 017, this instruction is treated as RESUME instead, which returns from an interrupt-service routine.

Instruction List: Extracodes	
READ	Moves the contents of the given I/O channel into A.
WRITE	Moves the contents of A into the given I/O channel.
RAND	Bitwise ANDs the contents of the given I/O channel into A.
WAND	Bitwise ANDs the contents of A into the given I/O channel.
ROR	Bitwise ORs the contents of the given I/O channel into A.
WOR	Bitwise ORs the contents of A into the given I/O channel.
RXOR	Bitwise XORs the contents of the given I/O channel into A.
EDRUPT	Meant for development and debugging only.
DCA	Double-precision CA; moves the contents of K,K+1 into the A,L register pair.
DCS	Double-precision CS; moves the 1's complement of K,K+1 into the A,L register pair.
QXCH	Exchanges the value in the Q register with the value at K.
AUG	Increments a positive value or decrements a negative one.
DIM	Decrements a positive value or increments a negative one.
DV	Divides the DP contents of A and L, concatenated, by the SP value in K. The quotient is stored in A and the remainder in L.
MP	Double-precision multiplies the value at A with the value at K and stores the result in the A,L register pair.
SU	Subtracts the value at K from the accumulator.
MSU	Treating the values in A and K as unsigned (assuming a leading zero), subtract them and store the 1's complement difference in A.
BZF	Jump to a location in fixed memory if the accumulator is zero.
BZMF	Jump to a location in fixed memory if the accumulator is zero or negative.
INDEX	Nearly identical to the basic version of INDEX, with the exception that it will never be interpreted as RESUME.

Values stored in memory on the AGC are most commonly stored as single-precision (SP) or double-precision (DP) values. An SP value consists of one leading sign bit and 14 magnitude bits; it is interpreted as a 1's complement value for arithmetic. A DP value consists of two concatenated SP values, with the less-significant one being of lower magnitude. DP values are also 1's complement, but their SP components can occasionally have inconsistent signs, something that the AGC has to be robust to. A small amount of unsigned arithmetic (this is often referred to as 2's complement, despite the fact that it cannot involve negative values) also occurs, for example in the inputs to the `MSU` instruction. However, most arithmetic instructions use 1's complement. For the purpose of divides and multiplies, values are treated

as being between -1 and 1. Overflows are handled differently than in most modern computing; a positive overflow wraps around to +0, while a negative overflow wraps around to -0.

2.3 Memory Map

The AGC has a very complicated memory layout. In general, it is broken up into read-write (usually called "erasable" and originally implemented as magnetic core memory) and read-only (usually called "fixed" and originally implemented as core rope memory) sections. However, each section is broken up into banks which can be mapped into address space using special registers. Erasable memory is broken into 8 256-word banks (E0-E8). Banks E0, E1, and E2 are permanently available at octal addresses 0000_8-1377_8, but another bank may be mapped into 1400_8-1777_8 by an appropriate setting of the `EB` register (note, however, that words at the very bottom of memory are actually registers). Similarly, fixed memory is broken into 36 1024-word banks (labeled 00-43 in octal notation). Banks 02 and 03 are permanently accessible at octal addresses $4000_8 - 7777_8$, and another bank may be mapped into $2000_8 - 3777_8$ by an appropriate setting of the `FB` register. However, because only 5 bits of `FB` are designated to affect memory mapping, and there are 36, not 32, fixed banks, a "superbank bit" (actually I/O channel 7) is used to select the upper 4 banks (the specific addressing scheme actually provides for 40 banks but 4 were missing on the physical AGC). Finally, the `BB` register mirrors the bank-selection bits of `EB` and `FB`, so that writing to it updates the corresponding bits on `EB` and `FB` and vice versa.

3 Microarchitectural Description

3.1 Stages

The processor's stages are represented as rules within an overarching module. Structurally, multiple rules are required because, as seen above, many instructions involve complex memory access patterns (such as `DAS`, which must read two words from memory, read two registers, perform a computation, write two words of memory, and write two registers). Because of similar structural concerns, and because of very complicated control hazards (for instance, `INDEX` can modify not just the target address but the opcode of the next instruction) and data hazards (any instruction that writes to memory can change the `EB`, `FB`, and `BB` registers, affecting the source of data and instruction loads) we did not attempt to pipeline our implementation. Hence no two rules in the processor run concurrently. FIFOs are used to transfer information between stages, although they never hold more than one element and essentially function as registers. Our stages are organized as follows:

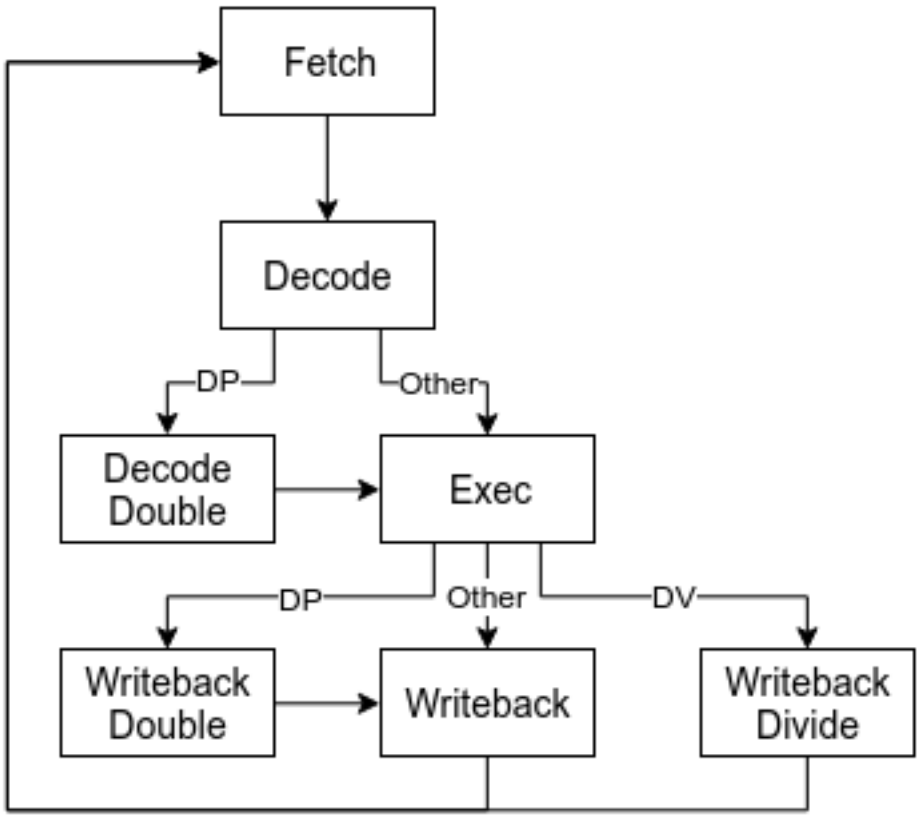

Figure 1: Basic diagram of processor stages.

The fetch stage combinationally reads the `Z` register (the program counter) and then sends a request to the instruction memory for the next instruction.
Decode catches the response. It handles interrupts if necessary (it is simpler to handle them here than in fetch). It then performs an `INDEX` addition if required by the previous instruction (using the state register `indexAddend`) and decodes the 15 bit instruction into a larger internal format. The first three bits are one portion of the opcode; the next two bits can also be used to distinguish between instructions, and the extracode flag from the previous instruction (using the state register `isExtended`) must be taken into account. Additionally, I/O instructions all have the same first three bits; three more identifying bits are needed to distinguish between them.
Once the correct logical instruction is identified, the decode stage does some preliminary work to prepare for execution. If memory, I/O channel, and/or register values will be needed for execution, the stage determines their addresses from the instruction and sends appropriate load requests to memory. The decoded instruction, along with flags indicating whether memory and register responses will need to be caught, is then passed to the next stage. Usually the next stage is execute; however, if a double precision instruction (such as `DXCH`) is being executed, the processor will first redirect to "DecodeDouble" to catch the first set of memory responses (the lower words) and send requests for the upper words, and only then continue on to execution.
The execution stage is the largest and most complex. Based on the given opcode, it deter-

mines and performs the correct operations on the any data received from memory. These operations can involve a significant amount of arithmetic. Most of the arithmetic operations performed, including multiplication (discussed in detail in section 3.2), are combinational; however, divide is implemented as a multistage circular pipeline (section 3.3).
After results have been computed by execute they are committed to memory, registers, and I/O channels. Most instructions proceed directly to the writeback stage as signals representing the values and target addresses of results. However, just as double precision instructions need an extra stage to fetch their extra words from memory before execution, they also need an extra stage ("WritebackDouble") to write their extra words (multiplication also requires this extra writeback stage). Writeback also sets the new `Z` register if it has not been explicitly modified by a memory or register write.
In the particular case of the divide instruction (`DIV`) it is necessary to catch the response of the divide module once division has finished in execution. Additionally, it is possible to make the specific double-precision write for this instruction in one cycle by taking advantage of the memory architecture. Thus a special rule ("WritebackDivide") is used in place of the standard writeback.

3.2 Multiplication

Of the arithmetic performed the most complex combinational operation is the multiply. This is done by splitting each argument into three five-bit unsigned values, which are multiplied, left-shifted appropriately, added, and then given the appropriate sign prior to returning, as shown below:

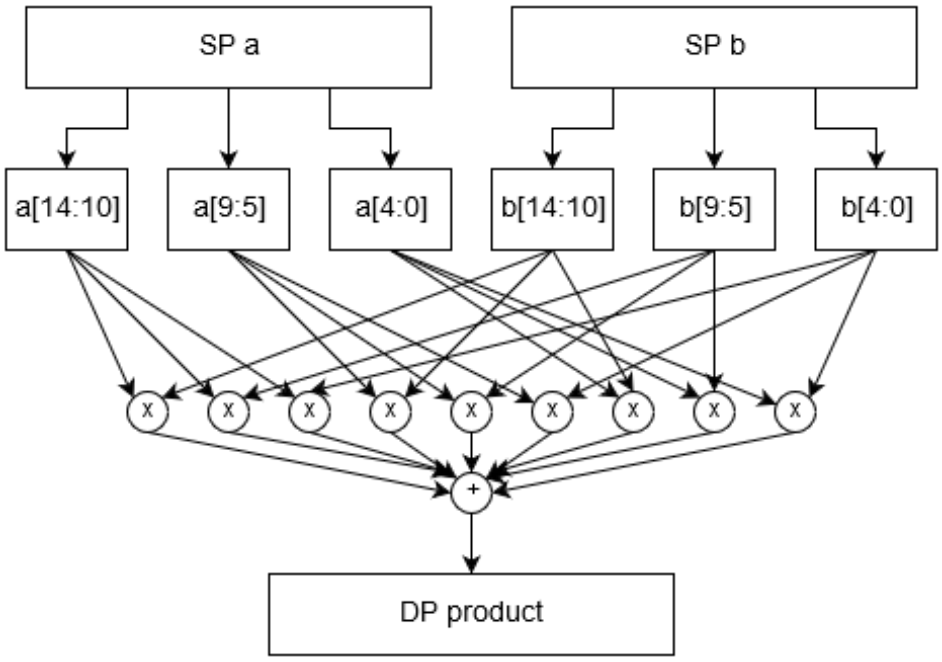

Figure 2: Description of multiplication.

3.3 Division

The sole non-combinational operation is performed by the division module, which performs long division on a double precision dividend and an single precision divisor, producing an exact quotient and remainder. Each cycle, it determines one more bit of the quotient; its general function on unsigned values is described by the figure below. The sign of the quotient is positive if the dividend and divisor have the same sign and negative otherwise, while the sign of the remainder will always match the sign of the dividend:

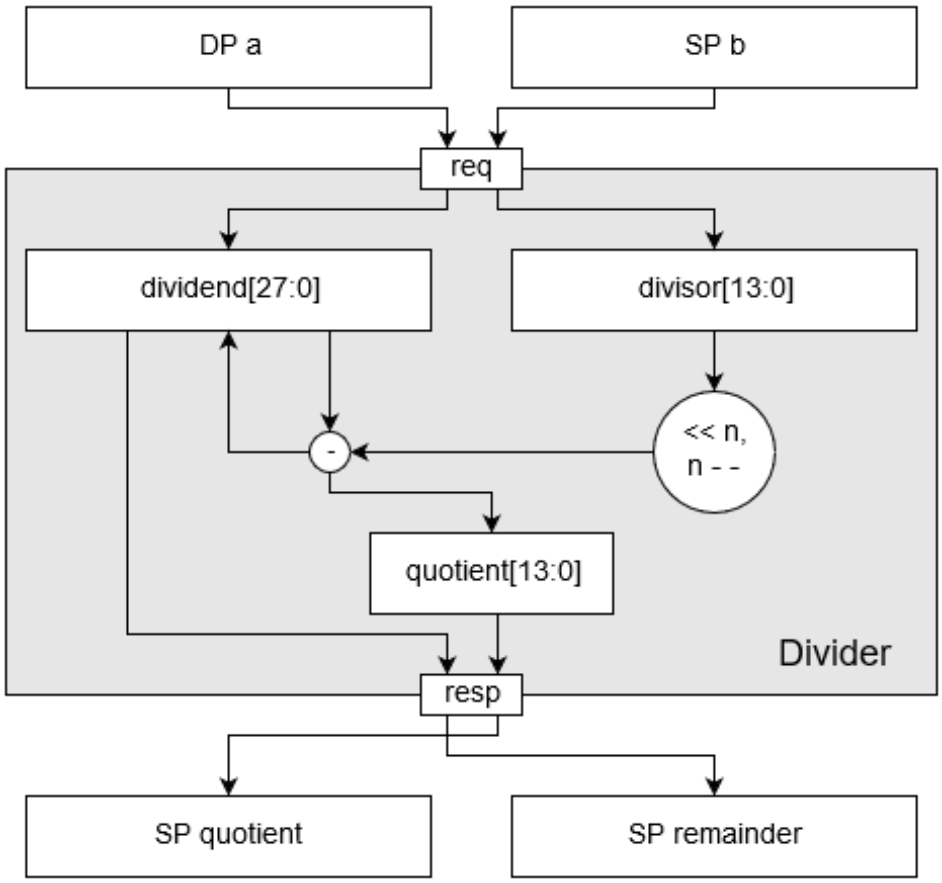

Figure 3: Description of divider module.

3.4 Memory and I/O

As discussed above (section 2.3) the AGC has relatively complicated and interconnected memory, registers, and I/O channels. All registers are mapped into the bottom of memory; the `EB`, `FB`, and `BB` registers control which banks of memory are accessible (and writing to one modifies another!); I/O channels 1 and 2 are aliases for the `L` and `Q` registers respectively; reading from the `CYR`, `SR`, `CYL`, and `EDOP` registers implicitly modifies them. Additionally, due to the many complex instructions, our memory module must service many requests at once: Fetch, Decode, and DecodeDouble can all request memory reads, their succeeding stages must catch the resulting responses, and Exec, Writeback, WritebackDouble, and WritebackDivide can all request memory writes. Finally, because we simulate the I/O channels using SceMi

(section 4.2) rather than actually connecting our FPGA to physical devices, we must buffer all I/O access so that I/O writes are visible to subsequent reads.
Thus, within our top level `mkAGC` module, we use two large modules, `mkAGCMemory` and `mkAGCIO`, to contain most of our state. `mkAGCMemory` maintains a large BRAM that holds both erasable and read-only memory, and a vector of registers. It performs memory address translation (mapping logical addresses to physical locations based on the contents of the bank selection registers as well as translating low memory addresses to register numbers) and exposes interfaces for instruction and data access (in simulation we used dual-port BRAMs, so it was convenient to dedicate one port for instructions and one for data, but the FPGAs we synthesized our design against do not seem compatible with dual-port BRAMs so this distinction is less meaningful). It also exposes several smaller functions like timers (which are just auto-incrementing registers). `mkAGCIO` is at its core a small secondary BRAM that buffers I/O channels. It then exposes internal facing interfaces for handling requests from the decode and writeback stages, and external facing interfaces that are effectively connected directly to SceMi. It communicates directly with `mkAGCMemory` to access state such as the `L` and `Q` registers and the superbank bit.

4 Testbench Architecture

4.1 Testing Considerations

Our goals when planning our testing were twofold: we wanted to confirm that our implementation of the AGC language was correct, and we wanted to try to run actual Apollo-era software on our design. The first goal in isolation suggests a relatively simple testing environment: build a design and write detailed unit tests that send a pass/fail signal over SceMi. However, the second goal requires much more complicated architecture. Not only do the subsidiary parts of our design, such as timing and interrupts, have to be correct, but we must also have some way of mocking the peripherals that the original software expects. Moreover, as long as we're running the actual software, it would be nice to try running an actual simulation of the Apollo capsule. These considerations led to the following design.

4.2 Testbench Architecture

The core of our testing architecture is a C++ testbench running on the host machine. On startup, it initializes the SceMi link and then uses it to transfer the target AGC binary to BRAM (in simulation we can also initialize the memory directly from a VMH file, but this lets us run as many programs as we want on the FPGA without regenerating the bitstream). Once finished, it waits for a TCP connection (port 19797) from third-party software that emulates the various peripherals and hardware (joysticks, gyroscope displays, etc.) surrounding the physical AGC, and once connected it then sends a start signal over SceMi to the AGC.[1] The testbench then monitors both connections (using separate threads) and translates data between the two formats (over SceMi we just transfer a one-byte channel number and one word of data, which needs to be converted to and from the special protocol

[1] http://www.ibiblio.org/apollo/yaTelemetry.html#LM_Simulator_by_Stephan_Hotto

used by LM Simulator). This gave us the flexibility to easily run a number of programs such as a pre-written validation suite.[2] We also successfully ran a number of custom programs, as discussed below.

5 Implementation Evaluation

Our implementation met all of the fundamental goals that we set out. Our AGC emulator successfully passes the validation suite, both in simulation and on the FPGA. In addition, we recently added interrupt and timer functionality (neither of which are explicitly tested in the validation suite). While we were unable to get the Luminary program running, largely due to the difficulty of debugging the processor while running self-modifying code, we did write a demo with a fraction of the functionality of Luminary (namely controlling spacecraft thrusters) that we will show during our presentation. Our final synthesized design meets timing at a 50MHz clock speed and takes between four and twenty cycles to complete an instruction for a total top speed of 12.5 million instructions per second; this shows more than a two-order of magnitude improvement over the original and a full 25% over our target. With regard to area, our design used 10.69% of slice LUTs, 2.95% of slice registers, and 4.46% of BRAM tiles.

We did end up with a somewhat different architecture than we initially planned, primarily because of the extra complexity presented by the double loads and stores. Getting a pipelined design to work in the time that we had was not very feasible. However, we did not ever have to fully overhaul our design.

We encountered significant challenges in moving our design from simulation to synthesis. Soon after we successfully passed the verification suite in simulation, we attempted to synthesize our design. Though it passed timing at 50MHz, we discovered that it did not pass the verification suite, and traced the problem to zeroes being read from one port of the BRAM. Due to a quirk of the Virtex FPGAs, it appears to be difficult to synthesize multiple-port BRAMs, as we were doing; similar issues to ours are known to occur when reading from multiple-port BRAMs with more than one clock. We resolved the problem by restructuring the way that we handled memory requests to work with a single port BRAM, after which our design synthesized and passed the verification suite.

In total, our implementation involved 3303 lines of Bluespec code in addition to supporting testbenches and assembly code for debugging.

6 Design Exploration

Given more time, there are several directions for design exploration that we would like to pursue. First, with a little more debugging, it might be possible to run Luminary and Colossus; this would ensure complete fidelity to the original AGC, as well as making an excellent demonstration.

Additionally, our design currently does not take full advantage of the ability of the FPGA to

[2] https://github.com/rburkey2005/virtualagc/tree/master/Validation

parallelize operations, as the processor is not pipelined. A two-stage pipeline would significantly improve instructions per second, and even if the processor itself is not pipelined, creating pipelined multiplication and division modules could substantially decrease the needed cycle time.

7 Conclusion

Thus, we successfully implemented a processor running Apollo Guidance Computer byte code in Bluespec, synthesized it for an FPGA, passed a thorough verification suite, and were able to use it to simulate flying an actual Apollo space capsule.