Apollo Guidance Computer: Computer Architecture Final Project

Lisa Hachmann & Anisha Nakagawa

November 2016

Abstract

We made a version of the Apollo Guidance Computer (AGC) that can run original apollo source code.

This historic computer features 16-bit instructions and memory, and an implementation of the original instruction set. We created a version of the Apollo Guidance Control with behavioral verilog that can implement this set of instructions. There were two versions of the AGC: block 1 was used in earlier unmanned missions and had 11 instructions, and block 2 was used in the manned missions to the moon and had 34 instructions, including the original set. In this project, we implemented all of the original block 1 instructions and three of the block 2 instructions, and we created tests to verify that they all behave correctly.

With these working instructions, we then used our AGC to run one of the actual files of source code in the Apollo missions. We focused on the instructions to calculate sine and cosine, which used the Hastings Approximation in calculation.

1 Background

The apollo guidance computer, along with its place in America's Space Race history, carries immense significance as one of the first computers to be built with integrated circuit logic. The AGC had many "quirks" about it, including its dramatic amount of memory for the addressable bits of data it had, making its design incredibly interesting. It used a lot of technology that was cutting-edge for the time and then evolved into the technology we have now. Also, it was designed very specifically for its task, which is interesting as a design note. In general, everyone knows what the AGC did and its role in history, which makes coding it incredibly fun.

To see more history and background on the apollo guidance computer, please see Presentation.pdf in our repository, listed below as the location of our code.

2 Implementation

We encoded the AGC in behavioral Verilog. We chose this over structural Verilog because although all of the schematics for the AGC are online, they consist of solely NOR gates- and 2100 of them. The AGC has a structure that includes mainly the memory (with fixed and erasable sections), program counter, a sequence generator that creates timing pulses, and registers for computations or specific memory.

2.1 Schematic

A general block diagram of the system is shown in 1. The main components of our verilog implementation were based on this layout.

The block diagram includes four central registers, which are used most often by the program. Most importantly, the accumulator (also called A register) is at the first address in erasable memory and is used to store the results of almost every computation. The Z register is also important, as it holds the value of the program counter, which is the address in memory of the current instruction. There are also many other named registers that are used to refer to different locations in erasable memory.

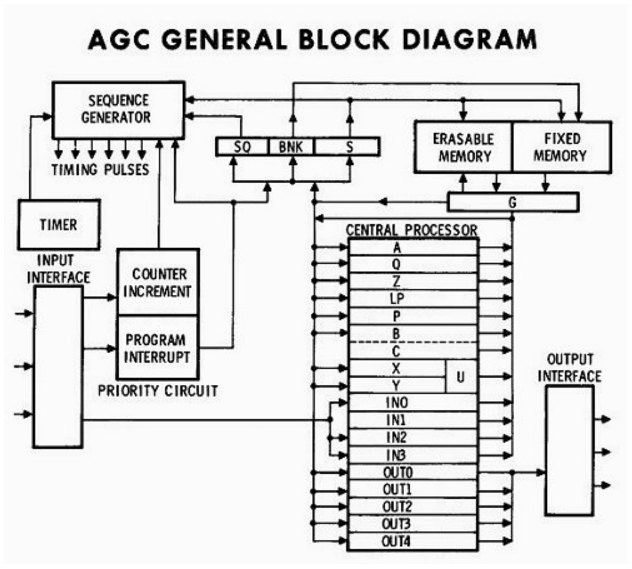

Figure 1: AGC block diagram

The control logic was all implemented with the timing pulses produced by the sequence generator. Each of the timing pulses is high one at a time, in a cycle. This is the same behavior we would observe from an n-stage ring counter made from D-flip-flops. The outputs from the sequence generator connect to the enable flags on the different components. When the timing pulse is high for one clock cycle, it sets the enable to the connected components to high at the appropriate time. The original apollo guidance computer used 12 different timing pulses to control the individual components, and used multiple cycles of the timing pulses while executing more complicated instructions. Since we create this in behavioral verilog, we needed far fewer timing pulses in our implementation. We only used 11 timing pulses, and were able to execute every instruction in only one cycle of timing pulses.

2.2 Instructions

The instructions for the AGC are encoded in 15 bits, where the first three bits are used to encode the Op Code (which type of instruction is being performed). The next two bits are the quarter code, which can be used to differentiate between two instructions with the same op code, and can also be used to specify which memory bank is being used. The final 12 or 10 bits (depending on the instruction) are the address in memory that the instruction is referring to. The address is commonly used to refer to a register that will be used in the given operation.

We built our project to be able to execute the original 11 block 1 instructions, and an additional 3 instructions from block 2. They were:

- Transfer Control (TC): Jumps to a different instruction
- Extracode (EXTRA): A special case of TC that sets the extracode flag to be high. The extracode flag is used to differentiate between different instructions with the same op code.
- Transfer to storage (TS): Saves the value in the accumulator into the specified address
- Count, Compare and Skip (CCS): A four-way branch that is dependent on the value in the register, and also performs a diminished absolute value calcuation
- INDEX: Add a constant to program counter, one version of jumping
- Clear and subtract (CS): Save the 1's complement value of a register into the accumulator
- Exchange (XCH): Exchange the contents between register A and Memory at the specified address

- MASK: Do a bit-wise AND operation between a register and the value in the accumulator
- ADD: Add the value of a register to the value in the accumulator
- MULTIPLY: Multiply the value of a register with the value in the accumulator, using double precision values
- DIVIDE: Multiply the value of a register with the value in the accumulator, using double precision values
- SUBTRACT: Subtract the value of a register with the value in the accumulator
- Transfer Control to Fixed (TCF): Jump to an address in fixed memory without saving the return address
- Double the Contents of A (DOUBLE): Double the contents of the accumulator and save back to it. This is a single precision operation
- Double Precision Double (DDOUBL): Double the double precision value stored in the A,L register pair, and store that value back into those registers.

In order to run our program, we had to convert all the instructions to binary.
To make instructions:

1. Either use specifically Block 1 instructions and 3-bit op. codes or specifically Block 2's or design a mixture of your own. Note that for Block 1, the quarter code and extracode flag were used to differentiate op. codes, while for Block 2 they also used peripheral codes.
2. Note that the instructions and operational codes are used with addition to the 10 or 12-bit address in memory "K". Thus an instruction with op. code 5 (no quarter code) would be (in octal) 50000 + K.

We verified that these instructions worked using a simple test bench, which is described in more detail in the basicmem.txt file in our repository. We stopped the program after each instruction to verify that the output (written to the registers) was as expected. A waveform of the test was also generated to confirm timing and to help with debugging. The waveform for the basic testing instructions is given in figure 2. Since the apollo guidance computer stored the values of the registers in locations in memory, it is nor possible to display the values in the waveform.

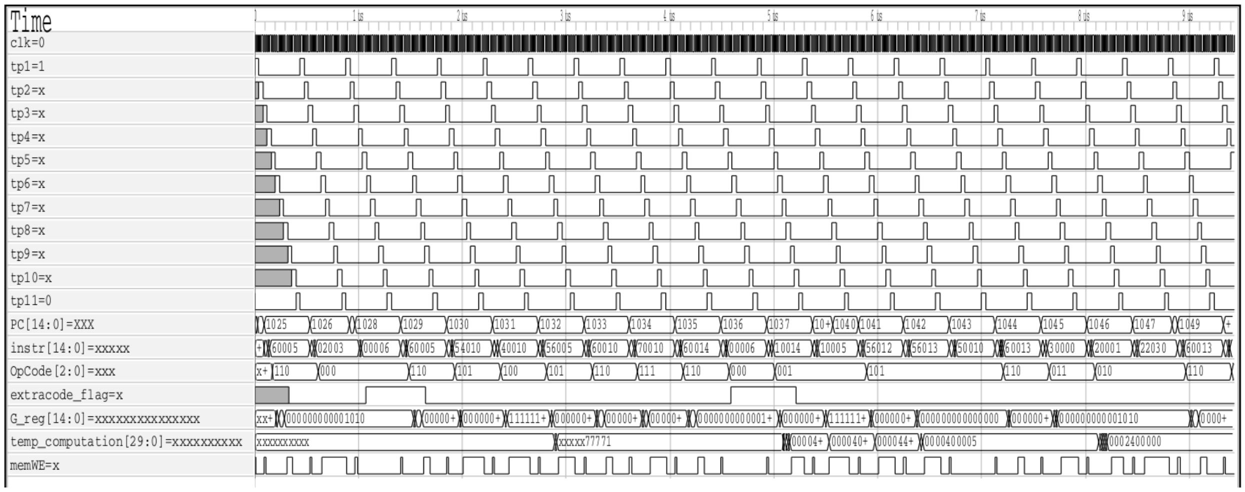

Figure 2: Waveform following the basic testing explained in basicmem.txt. This verified 13/14 instructions, excluding the multiply instruction, which was rigorously tested in the Hasting's approximation instruction set

2.3 Sine and Cosine

We chose to test our version of the apollo guidance computer using the original source code that calculates sine and cosine. We chose this section of assembly machine code because it used a limited set of the block 2 instructions and was easier to verify the results against expected values of sine and cosine. Since it is difficult to calculate sine and cosine with such a basic instruction set and limited memory, the apollo source code uses the Hastings approximation for these calculations. The Hastings approximation defines sine according to the following formula and constants:

$$sin(\frac{\pi}{2}x) = C_1x + C_3x^3 + C_5x^5 \tag{1}$$
$$C_1 = 1.5706268 \tag{2}$$
$$C_3 = -0.6432292 \tag{3}$$
$$C_5 = 0.0727102 \tag{4}$$
$$\tag{5}$$

We used the original source code instructions, which can be found in reference [8]. We encoded these instructions into the correct binary format for our program, and the waveform for these instructions is in figure 3. We ran our program with a few different input values to verify the sine calculations, the those results are in Table 1.

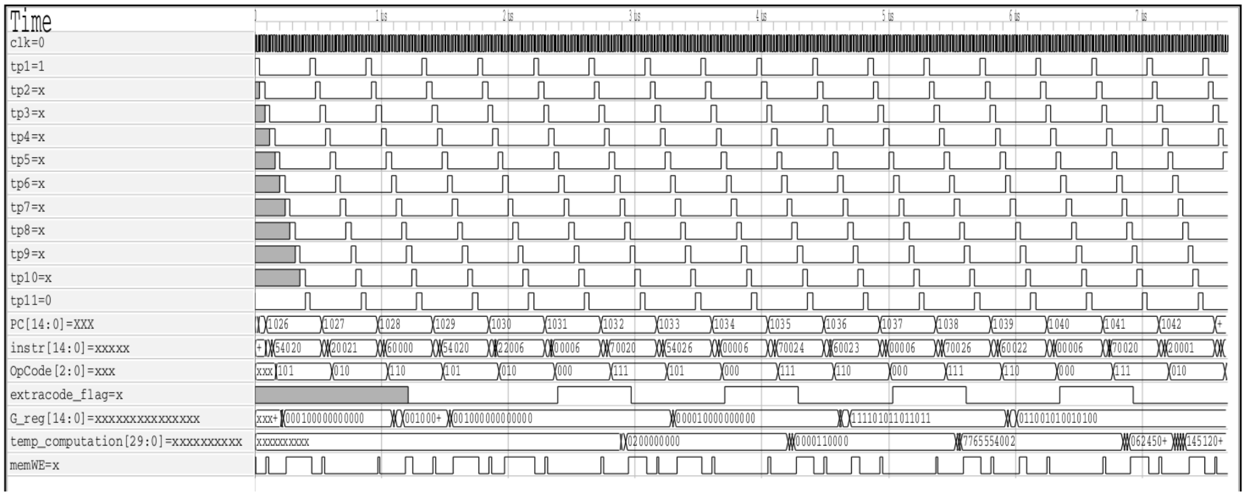

Figure 3: Waveform following the Hasting's Approximation for a sine wave, as explained in sincosbinary.txt. This verified that the AGC could approximately calculate single precision sine angles, and also tested the multiply instruction.

Table 1: Results of sine calculation from our program, compared to the actual value of sine

x	sin(x)	output sin(x) calculation
$\pi/4$	0.7071	0.8053
$\pi/8$	0.3827	0.3951
$\pi/16$	0.1951	0.1966
$3\pi/16$	0.5556	0.5974
$-\pi/8$	-0.3827	-0.4295

Our program was able to calculate sine according to the original source code, and our calculations are fairly close to the actual values of sine. The discrepancy could be attributed to two factors: first that the program uses Hastings approximation, and second that we are using single precision for this calculation.

With our setup, we truncate each of the intermediate calculations at 15 bits, because this is single precision. We speculate that the actual apollo machine had built in functionality for more accurate sine and cosine, however, there was not enough documentation to replicate this.

3 Execution Details

In order to build off our work, you will need our code and a version of memory. Ours is located in fullMem.dat, and needs to be filled in with instructions in order for the file to run. Sample instructions and preset constants are given in the file memoriginal.dat to encode basic instructions based off basicmem.txt or the file sincosoriginal.dat based off sincosbinary.txt. To run our code, copy one of the example instructions and paste into fullMem.dat before running. Out of 2047 lines of memory, we allocated 1023 for erasable and the rest as fixed memory, but that can be changed if all the memory addresses are changed consistently.

Our code is located on www.github.com/anishan/apollo_simulator.

Because the schematics for Block 1 and Block 2 are comprised of only NOR gates, the closest to a schematic that we followed is figure 1. Since our project used behavioral verilog, we mostly dealt with high level components without having to implement the smaller details. The designation of memory and the registers is up to the programmers, it just has to be consistent and large enough for the tasks encoded.

To run the AGC:

1. Make or use the file fullMem.dat, which has 2047 lines of 15-bit 0's.
2. Make instructions according to the steps enumerated in the previous section, and populate the instructions in the spot in fixed memory where the program counter starts
3. Include the line 15'b111111111111111 after your last instruction to finish the program
4. Pre-load any spots in memory with constants you need for computation
5. Run apollo.t.v with "iverilog -Wall -o apollo apollo.t.v" and then ./apollo Note that if the terminal does not display "Finished before the end of time", you should increase the delay in apollo.t.v.
6. Either look at fullMem.dat or the waveform produced (apollo.vcd) to verify your results in the registers in memory.
7. Remember that if you want to re-run the instructions with the same result, you need to clear your registers to their original states!!!

We wanted to use the original source code, so we used a chunk of the Single Precision Sine source code found on Apollo 11. It can be found here:
https://github.com/chrislgarry/Apollo-11/blob/master/Comanche055/SINGLE_PRECISION_SUBROUTINES.agc.

4 Reflections

List of Difficulties and "Gotchas":

- Hastings Approximation: Resources on Hasting's Approximation, used for sine/cosine, were hard to find. We were able to locate the original book by Cecil Hastings, titled *Approximations for Digital Computers*. It was also rather difficult to understand how the source code encoded this approximation.
- Timing Pulses: The documentation for which timing pulses control which actions was sparse, and we also did not have to account for hardware delays. Therefore, we created our own timing pulse sequence of controls.

- One's complement arithmetic: Learning one's complement's arithmetic had certain key differences to two's complement arithmetic that we were not used to.
- Double Precision Arithmetic: Using a mixture of single and double precision one's complement arithmetic and requiring truncation at some points in time lead to results that were much harder to interpret. Double Precision also includes only fractional numbers, leading to more confusing results.

List of possibilities to extend the depth of the project

- Division with negative numbers: Between fractions, double point arithmetic and one's complement, division was not tested for in all cases. A further extension of this project could verify division with negative numbers.
- The AGC had banks to address its large amounts of memory, which we did not encode. The banks for either Block 1 or Block 2 could be encoded to address memory.
- There are 23 more instructions (all Block 2) to potentially code into this AGC.
- The input/output interface to the astronauts (the DSKY) could be encoded to create an interactive AGC experience.

Overall, we feel that we learned an immense amount about not only the Apollo Guidance Computer, but about the technology around the time and how it has evolved. We learned many things about the design of the AGC that did not directly effect our code, but were still very interesting to learn, such as the physical format of memory with woven wire. It was fascinating to learn about the design decisions they made, especially when we knew enough background information to realize how clever some of their ideas were. We did have to make some assumptions about implementation when there was not enough documentation, but we had enough information to make reasonable abstractions.
We felt that the work plan was more accurate than in previous projects. We were able to front load most of the project as planned, and were successful in that. Surprisingly, the "general confusion" section did not need to be as long as it was, and we also did not have as many modules or test benches as expected. Verifying the instructions independently of test benches (through the waveform) took almost the same amount of time as the estimated test benches, so it did not affect our total time.

References

[1] Apollo Guidance and Navigation,
`https://www.ibiblio.org/apollo/hrst/archive/1717.pdf`

[2] Block I Apollo Guidance Computer (AGC): How to build one in your basement,
`http://klabs.org/history/build_agc/`

[3] Apollo Guidance System Documents,
`http://klabs.org/history/history_docs/mit_docs/index.htm`

[4] Block 1 Documentation,
`https://www.ibiblio.org/apollo/Block1.htm`

[5] Block II instruction format,
`https://www.ibiblio.org/apollo/assembly_language_manual.html`

[6] The Apollo Guidance Computer, Architecture and Operation, by Frank O'Brien

[7] Apollo 11 Github,
`https://github.com/chrislgarry/Apollo-11`

[8] Single Precision Sine and Cosine Instructions
`https://github.com/chrislgarry/Apollo-11/blob/master/Comanche055/SINGLE_PRECISION_SUBROUTINES.agc`