CS2253 Programming Project 2

March 10, 2022

Final deliverable due 11:59 pm on the last day of classes (April 13). Earlier deliverable on March 28. Individual work.

1 Introduction

The Control Data 1604 computer was historically important as one of the earliest transistorized computers, and one of the early systems of the famed supercomputer designer Seymour Cray. The design is 4 to 5 years older than your professor, and during the early 1960s, over 50 of them were sold at a cost of more than $10 million each in today's money. (This was a commercial success.) The machine had 0.00192 GB of main memory and executed instructions about 100 000 times slower than a current desktop. See `https://en.wikipedia.org/wiki/CDC_1604` for an overview of the 1604.

In this project, you will use C to write a simulator for a substantial subset of the 1604. Given the speed of the computer running your simulator, your simulator will probably execute programs much faster than the real machine ever could.

To keep the project manageable, you will ignore the machine's IO and interrupt capabilities, and you can take some liberties with floating-point calculations.

Your tools are to be written in C18 and compile without warnings when compiled in the CS lab with gcc using command line options

```
-std=gnu18 -fsanitize=undefined -fsanitize=address -Wall
```

Important Note: If you do development on a different platform and/or with a different compiler version, you can expect your program will require adjustments whose time cannot easily be predicted. C is not Java. I might not accept programs that require some other platform or compiler, or I may apply a substantial penalty. Many C programs may appear to work correctly without the sanitizers but still have nasty bugs or potential bugs. "But it seems to work perfectly on my platform" is not good enough.

If you want to develop with a different platform or compiler, you should **frequently** make sure you can still compile and run on the lab machines. Otherwise you will end up submitting what is, from my viewpoint, a non-working program. It would be nice if 2022 were the first year this did not happen to many of the people in class.

The next section contains details of the required programs.

2 Details

As a reference to the 1604, please see `http://bitsavers.org/pdf/cdc/1604/018c_CDC1604_Manual.pdf` You may prefer to consult the manual for a slightly changed version (for the 1604A) [1] because it has flowcharts of some of the operations you need to implement.

In order to write a simulator, you don't need to understand the microarchitecture, but you do need to understand the ISA. Unfortunately, when the manual was written in 1959, people had not yet made a clear distinction between microarchitecture and ISA. Also, some of today's standard terminology had not yet been adopted. In the manual, I think you will be able to skip over pages I-4, II-45 to II-48, III-2 to III-14, III-24 to III-25, the Control Sequence discussion on III-32 and the Storage Test Keys on IV-3. There is a long discussion of the obsolete kind of memory used on pages III-15 to III-21 which you don't need to read to write the simulator, but which might be interesting to anyone interested in computer history[2].

I recommend that you keep the glossary (pages 4 and 5 of the document) on hand as you try to read the 1604 documentation. In particular, I found the discussion of "full exit" and "half exit" confusing before I consulted the glossary. "Logical product" or "logical multiplication" is an old-fashioned way of describing the AND operation.

The way subroutines are called and returned is weird and involves self-modifying code, where the bits for the return address overwrite some bits in a (jump-type) instruction positioned just before the start of the subroutine. So when the subroutine is ready to return, it can jump to the address just before the subroutine, which will be a jump instruction whose overwritten address returns you to the original call site. Some early microprocessors from the 1970s also had (different) weird ways of doing calls and returns.

If you check the photo at on page 3, you can see the human operator seated in front of a weird thing. That is the operator's *console*, which displays the current contents of each register in binary and octal, and has switches to allow registers to be changed. It also has several "lever keys" that the operator can press and that some instructions can inspect. You can ignore the "storage test keys" discussed on page IV-2, but there is also a stop-start key that can be used

[1] `http://bitsavers.org/pdf/cdc/1604/245a_1604A_RefMan_May63.pdf`

[2] The "core memory" *was* actually implemented with a tiny magnetic bead (a "core") for each bit of memory, with wires threaded through each bead. It became obsolete with the invention of semiconductor RAM chips around 1970, but the name lingers on in the form of "core dump" in Unix/Linux, when an aborting program writes out a "core file" of its memory contents for postmortem debugging. Also the logo of the Canadian Information Processing Society (CIPS) is based on a picture of a core bead with wires. See `https://cips.ca/`.

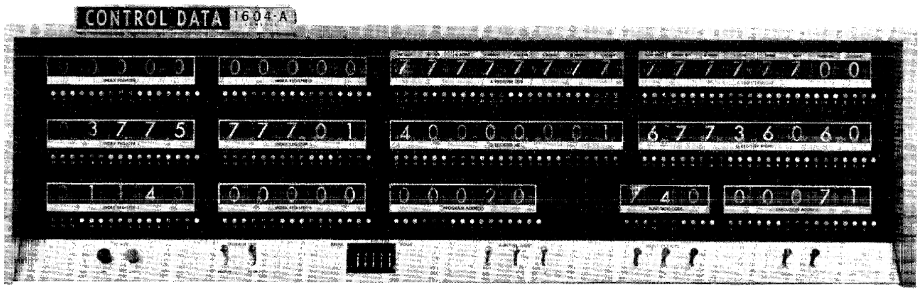

Figure 1: The console of a 1604A computer. (The 1604A is some minor modification of the 1604.)

to run the program at full speed or can be used to single-step the computer. Your simulator should provide the same kind of information displays that the 1602 console provided.

Another CDC document shows the console more closely (see Fig 1). When your simulator is doing single steps, your program should display information in the same layout as the actual console, in terms of the arrangement of octal numbers. I can't quite make out all the labels on the console, but the index registers are on the left. Top to bottom, the leftmost are index registers 1, 2 and 3. To the right of them are index registers 4, 5 and 6 (top to bottom). The program counter is in the bottom centre (showing 00020). The A register is on the right top (the left 24 bits are shown as 77777777 and the right 24 bits are shown as 77777700). Beneath the A register, we see the Q register (its left bits are 40000001 and its right bits are 67736060). The rightmost set of digits on the bottom displays the least significant 15 bits of the current instruction's machine code, which is called the "execution address". You can ignore the set of digits on the bottom, to the left of the execution address display.

2.1 1604 instruction subset

You should ignore conditions that cause faults, as well as instructions related to input and interrupt. In particular, the INT, OUT and EXF instructions will not be simulated.

3 Simulator

We need some way to get an initial program and some initial data into memory. In the 1604, this would probably have involved using the paper-tape reader, which we are not simulating. Instead, your simulator program will start by prompting the user to provide the name of an "octal file" (whose format will be described soon) that will be loaded into memory. After the first octal file has been loaded, your simulator should prompt to see if there are any more octal files to be loaded, and if so, get their names and load them.

The simulator will then ask you to provide the initial value for the program counter, and then it will proceed to fetch and execute instructions in single-step mode. As it fetches and executes instructions, either in single-step mode or running full speed, it should check (using "nonblocking input" when running full speed) whether certain keyboard keys (g, G, e, E, u, U) are pressed. There are some additional keys that can be pressed only when the computer is in single-step mode. The total list of operations is below and each corresponds to something that the operator could do on the 1604 console.

g: put the start/step lever key down

G: put the start/step lever key up

e: put the first selective stop lever key down

E: put the first selective stop lever key up

u: put the first selective jump lever key down

U: put the first selective jump lever key up

b: set the breakpoint (prompt for a 5-digit octal breakpoint address). The initial breakpoint value is 0.

m: display memory contents (prompt for a 5-digit octal address and then print the contents of this address and the next 3 addresses in octal)

d: dump memory contents (prompt for a 5-digit octal address and then print a "memory dump" into a file called "core.oct", in a format described below).

s: set memory contents (prompt for a 5-digit octal address and a 16-digit octal data value to be stored in the address)

a: set the A register (prompt for a 16-digit octal value)

q: set the Q register (prompt for a 16-digit octal value)

p: set the program counter (prompt for a 5-digit octal address)

1..6: set the corresponding index register (prompt for a 5-digit octal value)

You don't need to worry about the second and third selective stop or selective jump lever switches.

3.1 Core.oct file format

The memory dump should consist 50 lines, each starting with a 5-digit octal number (an address A), then having a tab, then four 16-digit octal numbers separated by spaces. These four numbers give the memory contents at addresses A, $A+1$, $A+2$ and $A+3$. The address for the first line is whatever the user specified. The address of each subsequent line is 4 larger than the address of the previous line.

3.2 Octal file format

The octal file (memory input) starts with a line that indicates the address where the remaining contents should be loaded in memory, just like the LC3tools .bin file, except that the number is given as 5 octal digits.

Remaining lines of the octal file can be blank lines or *ordinary* lines or *comment* lines.

- Ordinary lines start with some whitespace, then have a sequence of 16 octal digits. Digits can be contiguous or have whitespace between them. After the octal digits, there is an optional semicolon, which means the rest of the line is a comment. If there is no comment, then the rest of the line must be whitespace.
- Comment lines start with some whitespace, then have a ; marking the beginning of a comment that continues until the end of the line.
- Blank lines can be empty or have some whitespace characters.

3.3 Writing a simulator

There are a variety of fancy high-performance ways of writing a simulator that you might discover from Dr. Google. They will confuse you and make life difficult. We don't care nearly as much about simulator performance as about your sanity and frustration level. So, your C program can use a big array to simulate memory. You can have a `int16_t` variable that represents the current value of the PC, and you can handle the 5 index registers register similarly. The 48-bit A and Q registers can be represented by variables of type `int64_t`. A complexity is that the 1602 uses 15-bit values for indexes and the PC, and you will have to handle the use of 1s complement vs 2s complement. The 48-bit floating-point format does not correspond to any of the IEEE-754 formats, but if you do bit-masking magic you should be able to take a 48-bit floating point value, form the equivalent C double, carry out the math on doubles, then use bit-masking magic to convert the result back to 48-bit format. To make this easier, you can assume there are no overflows or underflows.

You can write the fetch-execute cycle as an infinite loop whose body is very large switch statement on opcode. One 1604 complexity is that one fetch brings in a pair of 24-bit instructions, an "upper" and a "lower" instruction that would normally be executed in that order before a fetch of the next 48 bits is done.

3.4 Test programs

We need some hand-generated octal files for simulator testing. Parts of this task will be divided between members of the class and I plan to provide a few also. You are probably about 60 years younger than the last people to seriously program the 1604, so I would suggest that you not list 1604 machine-code programming as a skill on your resume, unless you want to impress your grandparents or great-grandparents.

Multiply by adding: Everyone ought to code a version of the "multiply by adding" (textbook Fig 4.7) program, which then continues on to compare the result to a multiplication instruction's output, putting 1 or 0 in octal address 3000, depending whether the results match. The inputs will come from addresses 3001 and 3002 (octal).

Adding numbers from memory: Your second machine-code program should resemble the program in textbook Fig 5.14, in that it adds up the numbers in a bunch of memory locations using a loop. For the 1604 version, use a memory addressing mode that uses index register 2, and increment this register in the loop. The final sum should go in memory address 4000 octal, and the input should come from addresses 4001, 4002, etc. Use -0 (a lovely quirk of 1s complement) as the sentinel.

Masked search: Use the MTH instruction to make a program that searches in a string of length 8 for a character whose ASCII code is larger than the ASCII code in the A register. The string in question is stored in memory right after your program and its value is 'COMPUTER', with one ASCII character per memory address. However, each ASCII code occupies the least significant 7 bits of that memory location, and the most significant 41 bits contain irrelevant information. In your program, you can put any non-zero value into these 41 bits. If there is a bigger letter, the memory location immediately after the R should store 1, otherwise it should store 0. If you run this code, test it with the A register containing the ASCII code for 'S' and re-test it with the ASCII code for 'V'.

The 1604 predates ASCII, but don't worry about that.

4 Deadlines

Deliverables are expected throughout the next few week, on D2L.

- First deliverable (due March 28, 11:59 pm): The test programs. (You should have a start on your basic simulator by this time too, or you will have a rough couple of weeks ahead of you.)

- Final deliverables (due April 13, 11:59 pm; may not be late): The simulator. If your program needs to be compiled differently than the specified way, give the details.

5 Coöperating

Sharing C source code with others is cheating or plagiarism. The same is true for the test programs in the first deliverable. If you care to develop more test programs for your debugging, you are free to share these around.

Note that you might be able to find source code for existing tools for processors similar to the 1604. Do not try to reuse their code — first, that would be plagiarism; second, they probably are doing sophisticated things that will confuse you horribly.

6 Grading

The basic simulator consists of the ability to load an octal file into memory, set the PC and a breakpoint, run at full speed until the breakpoint is hit, and do a memory dump at a user-specified location.

Instructions simulated should be everything except INT, OUT, EXF, FAD, FSB, FMU, FDV, SCA, SCQ, MUF, DVF, EQS, THS, MEQ, MTH. You do not need to handle indirect addressing.

The full simulator adds the handling of the noted console-related commands, indirect addressing, and all instructions except INT, OUT, EXF.

Subtask	Value
Octal files in first deliverable	20
Basic simulator	55
Full simulator extras	25

7 Submitting

Please submit on D2L.