# feilipu

Stuff I need to write down.

# Characterising Am9511A-1 APU

Posted on February 22, 2017 by feilipu
I've built a Z180 based board (https://feilipu.me/2016/05/23/another-z80-project/), supporting the AM9511A APU. Partly for historical enjoyment. Partly because the AM9511A-1 is actually still faster at arithmetic than "modern" Z80 devices.

For example, from the Am9511/Am9512 Floating Point Processor Manual (ftp://bitsavers.informatik.uni-stuttgart.de/pdf/amd/_dataSheets/Am9511A-9512FP_Processor_Manual.pdf) by Steven Cheng, we have comparison tables. On average the Am9511A APU (at 1.966MHz) produces a hardware floating point divide in 165.9 cycles (of a 2MHz 8080 processor). Converted to my Am9511A implementation (at 2.304MHz), we have the equivalent in 141.5 cycles of the 2MHz 8080. Converted to best case modern Z180 terms (overclocked to 36.864MHz) this is 2,609 CPU cycles to return a hardware floating point divide.

To produce an equivalent software floating point divide, using the equivalent vintage LLL floating point library, requires 13,080 cycles.

This means that floating point on the 40 year old AM9511A-1 APU is still 5.0 times faster than an overclocked Z180 running antique 8080 code. Sweet!

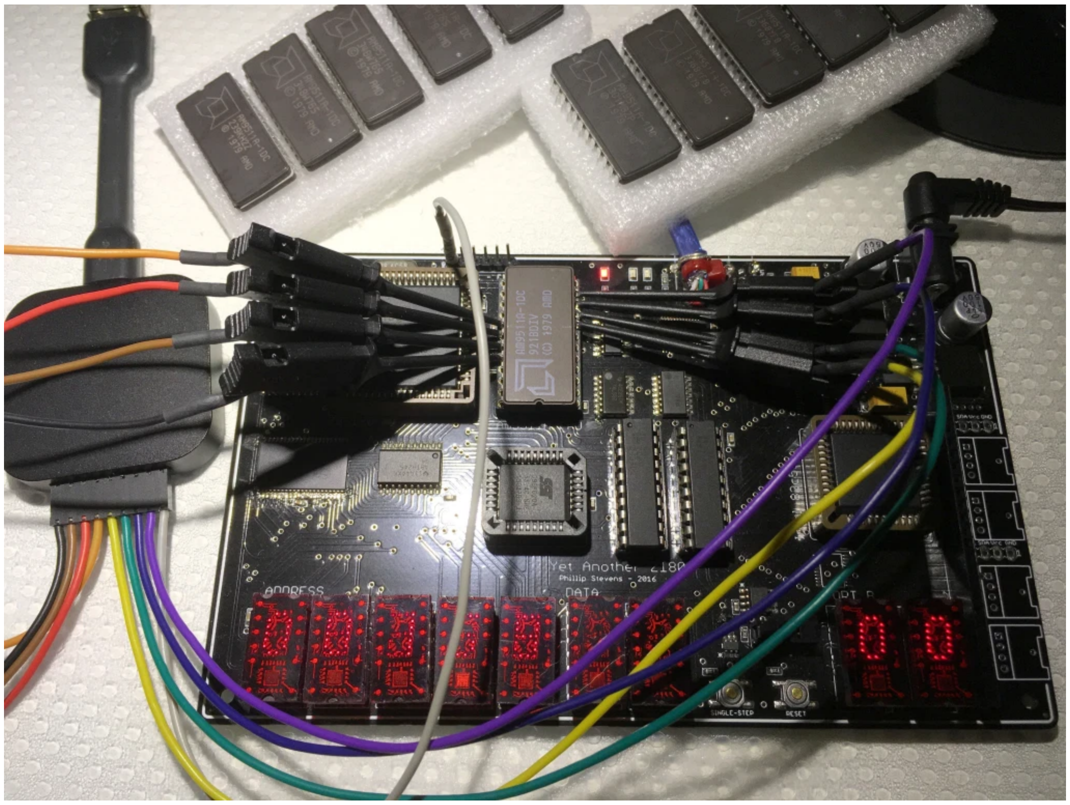
## Testing

I'm integrating an Am9511A-1 APU device into my YAZ180 build. The basic device is capable of operating at 3MHz. But, I've found that driving one sample at 3.072MHz doesn't work. But, it works fine at 1.536MHz, and at 2.304MHz.

This one example has 83.33ns delay between RD or WR and PAUSE signal being operated. This means that it should comfortably operate with the minimum of one wait state when the Z8S180 is running with a 18.432MHz bus.

And, I've got to say that these devices run hot… OHS issue hot. There is a reason they are provided in a ceramic package. They sink 70mA at 12V plus another 70mA at 5V, and all that energy has to go somewhere.

Originally, I had created the timing for the Am9511A-1 is generated by dividing the Z8S180 18.432MHz system clock by 6. The divide FCPU by 6 for FAPU CLK 3.072MHz is done with a SN74LS92N device. And for test purposes, the Z8S180 is also half rate clocked at 9.216 MHz, producing 1.576MHz for the APU.



YAZ180 Test Rig – Am9511A-1

## Initial Testing

The test is initially pretty simple, really just a proof of life. Will they properly push and pop data at 3.072MHz?

If not, then I'll need to redesign the YAZ180 to operate the Am9511A-1 at a lower APU clock. But, if my initial samples are just not the up to specification, then they can be secondary devices.

Ok, now what kind of devices have I got to hand, and lets see what the results are…

| Front Serial ID | Country & Rear ID | 3.072MHz | 1.576MHz |
|---|---|---|---|
| **239KH2Z** | Malaysia 9237FP | Fail | Pass |
| **301MBZP** | Malaysia 9252CP | Fail | Pass |
| **3368YF8** | Malaysia 9335CP | Fail | Pass |
| **348W76S** | Malaysia 9347DP | Fail | Pass |
| **921BDIV** | Philippines 8917WM | Fail | Pass |

Kind of boring really. Pretty clear that AMD oversold the capability of its Am9511A-1 to run at 3MHz. Or, I'm not feeding them with the correct timing.

I'll need to redesign the YAZ180v2 (https://feilipu.me/2017/02/26/yet-another-z180-v2/) to provide a divided by 8 clock at 2.304MHz, and cross my fingers that they work at that clock rate.

## Update September 2017

Following a substantial investment in testing of the solution in April and August, I've finally got a working system that can reliably use the Am9511A-1 to produce computational results.

The main issue was described in the Retro Challenge (https://feilipu.me/2017/02/26/yet-another-z180-v2/) period. The Z180 CPU I/O timing is slightly different to the Z80, in that it doesn't provide any interval between the WR/RD deselection and the CS deselection. For most peripheral devices this isn't a problem, but for some it causes them not to work. Hitachi recognised this problem, and provided a shortened I/O signal called the "E Clock", which can be gated together with the WR/RD signals to gain the timing desired.

The Am9511A doesn't require the RD signal to rise before the CS signal. Its TRCS time is 0ns. However it does require the WR signal to rise 60ns (or for Am9511A-1 30ns) before the CS signal. This can be achieved by gating the WR signal together with the E Clock signal. The Z180 E Clock is 1/2 a Phi cycle shorter than the WR signal during I/O instructions.  Therefore provided that the Z180 Phi is slow enough, then the required timing can be held. At 18,432MHz, the Z180 Phi signal is 54ns long, and therefore 1/2 Phi is 27ns, which is too short to work consistently. However, the Z180 can halve its Phi speed easily, and simply by changing the CMR register the Phi clock can be reduced to 9.216MHz, which in turn gives us (theoretically) 54ns TWCS for the Am9511A-1.

In practice gating the WR signal through the E Clock provides 41ns for TWCS, which is enough to work reliably with Am9511A-1 devices.

However, as the Am9511A-1 timing is derived from the Z180 clock (divided by 8), running Phi at 9.216MHz means that the APU is in turn under clocked to 1.152MHz. We can avoid this waste of capability, by managing the Z180 clock precisely during the command interrupt generated by the Am9511A-1. On entry to the interrrupt, we reduce the Phi to allow commands (TWCS) to work, and on exit from the interrupt we return the Phi to its original setting of 18.432MHz (and the APU to 2.304MHz). This also conveniently avoids having to modify the Z180 ASCI code to manage serial communications in the light of the reduced system clock.

I have also tested the Am9511A-1 using this revised WR to CS timing, with my original clocking of divide by 6 from Z180 system clock, or 3.072MHz, and it doesn't work. This means that we actually do need to stay within the specification, and not overclock the APU.

## z88dk Driver

The z88dk supports the YAZ180 platform, so I have added a driver for the Am9511A into the device folders (https://github.com/z88dk/z88dk/tree/master/libsrc/_DEVELOPMENT/target/yaz180/device/am9511a). The driver is based on an interrupt driven model. As noted in the Am9511A Processor Manual (https://ia601604.us.archive.org/27/items/bitsavers_amddataShecessorManual_4528121/Am9511A-9512FP_Processor_Manual.pdf), there are 4 different models that can be adopted.

The Am9511A Pause signal is connected to the Z180 Wait signal, so if the CPU generates a request that requires time to fulfil, then the APU will cause the CPU to Wait by signalling Pause. Clearly this means that the CPU cannot do anything while it is in Wait mode. This model of working is called Demand Wait, and it provides the fastest APU response, but doesn't support using the CPU effectively.

The APU status register can be read without the APU causing a Pause response. Therefore the CPU can poll the APU by reading the status register. In this situation the CPU can continue profitably, but response to APU requirements (for new operands, or commands) is limited to the polling rate. As different APU commands can be completed in different times, the optimal polling time can be difficult to calculate.

The APU End signal can be connected to a CPU interrupt, and in the case of the YAZ180 it is connected to the NMI, which is then triggered at the end of each command, when the APU is ready to receive a new command. If the APU commands are buffered in advance, then using this interrupt mechanism the APU can complete a long sentence of commands "autonomously" and in parallel to the CPU activities, interrupting only when it needs to load a new command. This is the mechanism used in the z88dk driver (https://github.com/z88dk/z88dk/tree/master/libsrc/_DEVELOPMENT/target/yaz180/device/am9511a).

Some ideas for creating an optimal interrupt driven driver for the Am9511A were inspired by reading "An Efficient Software Driver for Am9511 Arithmetic Processor Implementation", B. Furht and P. Lee, 1984.

The final driver option is to connect the APU Service Request signals to the CPU DMA interface. In the YAZ180 these hardware signals are connected, and if desired a DMA enabled driver could be written to take advantage of this interface. However as only 4 bytes of information can be transferred in each load or unload command, at face value there seems to be little advantage in building a DMA software interface.

The C interface for the Am9511A driver supports direct access to the Am9511A, like the asm driver, and hopefully it can be integrated into the z88dk math library options, to support transparent usage of the APU where it exists.

Currently, some simple C interface code is now available on z88dk (https://github.com/z88dk/z88dk/blob/master/libsrc/_DEVELOPMENT/target/yaz180/device/am9511a/readme.md), which allows us to do performance testing.

## Performance Testing

A simple test of seeking prime numbers exercises the floating point divide and 32 bit fixed point divide. This "brute force" method is not optimised for the APU as 4x 32 bit numbers must be loaded into the APU, but only two divides and a subtraction are done, for each calculation cycle. During the operand loading process, the CPU must also be slowed down to half rate, but since this is during an non-maskable interrupt there is little effect on the overall system.

So, some comparisons. First using Nascom Basic, we have a baseline, for seeking 1000 prime numbers.

```
20 PRINT "LIMIT";
30 INPUT L
40 FOR N = 3 TO L
50    FOR D = 2 TO (N-1)
60      IF N/D=INT(N/D) THEN GOTO 100
70    NEXT D
80    PRINT N;
90    GOTO 110
100   PRINT ".";
110 NEXT N
120 END

200 REM 124.7 Seconds (hand timed) - Z180 36.864MHz - Nascom Basic
```

Adding the Am9511A to the calculation, by doing the inner loop in assembly and calling the APU for the divide and subtract duties, does speed things up.

```
20  PRINT "LIMIT";
30  INPUT L
40  FOR N = 3 TO L
50    IF USR(N) = 0 THEN GOTO 100
80    PRINT N;
90    GOTO 110
100   PRINT ".";
110 NEXT N
120 END

200 REM 90.4 Seconds (hand timed) - Z180 36.864MHz - Nascom Basic
```

This is interesting, as the Am9511A is running at 2.304MHz (straight out of 1977), yet it is STILL faster than software on a 36.864MHz Z180.

Now, what happens when using the z88dk, to do the calculation in C?

```
void main(void)
{
  static signed long l, n, d;
  printf("Limit: ");
  scanf("%ld", &l);
  for (n = 3; n != l; ++n)
  {
    for (d = 2; d != (n-1); ++d)
      if ((float)n/(float)d == n/d) break;
    if (d == (n-1))
    printf("%ld", n);
      else
    printf(".");
  }
}

// 67.8 Seconds (hand timed) - Z180 36.864MHz - z88dk sdcc_iy
// 62.7 Seconds (hand timed) - Z180 36.864MHz - z88dk newlib

// zcc +yaz180 -subtype=basic_dcio -vn -SO3 -lm -clib=sdcc_iy --max-allocs-per-node200000 prir
// zcc +yaz180 -subtype=basic_dcio -vn -SO3 -lm -clib=new primesC.c -o primesC -create-app
```

Using the assembly math routines in z88dk is now the fastest solution. But, we'd expect C to be faster than Basic. What happens if we do the divides and subtractions in the Am9511A. That's why we're here.

```
void main(void)
{
  static signed long l, n, d, r;

  apu_reset( (void *) 0x2021 ); //INITIALISE THE APU with the NMI VECTOR ADDRESS

  printf("Limit: ");
  scanf("%ld", &l);

  for (n = 3; n != l; ++n)
  {
    for (d = 2; d != (n-1); ++d)
    {
      apu_cmd_ld( &n,   APU_OP_ENT32);
      apu_cmd_ld( NULL, APU_OP_FLTD);
      apu_cmd_ld( &d,   APU_OP_ENT32);
      apu_cmd_ld( NULL, APU_OP_FLTD);
      apu_cmd_ld( NULL, APU_OP_FDIV);
      apu_cmd_ld( &n,   APU_OP_ENT32);
      apu_cmd_ld( &d,   APU_OP_ENT32);
      apu_cmd_ld( NULL, APU_OP_DDIV);
      apu_cmd_ld( NULL, APU_OP_FLTD);
      apu_cmd_ld( NULL, APU_OP_FSUB);
      apu_cmd_ld( &r,   APU_OP_REM32);

      apu_isr();         // calls the ISR to trigger the process
      apu_chk_idle();    // blocks until the APU is idle

      if (r == 0) break;
    }
    if (d == (n-1))
      printf("%ld", n);
    else
      printf(".");
  }
}

// 94.9 Seconds (hand timed) - Z180 36.864MHz - z88dk

// zcc +yaz180 -subtype=basic_dcio -vn -m -SO3 -clib=sdcc_iy --max-allocs-per-node200000 @prir
// appmake +glue -b primesC_APU --ihex --clean
```

Well the end result is (surprisingly) similar to using the Am9511A together with Basic. The issue is that it takes a long time to load and unload the operands to the APU, and to process (cast) them, and then we only do a relatively simple operation when they're there. Relatively speaking for this test, we're bound by the I/O rate of the APU, which is quite slow, and this does not demonstrate actual the compute rate of the APU.

```
APU_OP_ENT32
APU_OP_FLTD -  56-342 Cycles
APU_OP_ENT32
APU_OP_FLTD -  56-342 Cycles
APU_OP_FDIV - 154-184 Cycles
APU_OP_ENT32
APU_OP_ENT32
APU_OP_DDIV - 196-210 Cycles
APU_OP_FLTD -  56-342 Cycles
APU_OP_FSUB -  70-370 Cycles
APU_OP_REM32
```

However, this is still a good result. We're comparing a modern overclocked z180 at 36.864MHz, with an ancient device, released in 1977 (http://www.cpushack.com/2010/09/23/arithmetic-processors-then-and-now/), running at 2.304MHz. And, we have achieved comparable results.

## Update July 2018

Over the past year I've done quite a lot with the Am9511A. One interesting application was using it to calculate a Mandelbrot set, and generate a text image.

From the point of view of drivers, I re-wrote the interrupt based driver (https://github.com/feilipu/yaz180/blob/master/yabios/asm_common1_driver.asm#L900) to use a 3 byte operand address in the ring buffer (https://feilipu.me/2018/05/28/three-rings-for-the-z80/), to support access to the full memory space of the Z180.

I also uncovered a hardware bug (https://github.com/feilipu/yaz180/blob/master/construction_notes.md), which was caused because I was resetting the APU clock divider with the system reset. This meant that the APU was not getting 5 clock cycles with Reset held high, and would therefor not reset reliably. Removing the reset signal from the divider chip solved this issue.

This entry was tagged am9511a, amd, apu, floating point processor, floating point unit, fpu, hardware, retrocomputing, zilog. Bookmark the permalink.

## 3 thoughts on "Characterising Am9511A-1 APU"

1. Pingback: Yet Another Z80 Project (YAZ180) | feilipu
2. Pingback: Yet Another Z180 (YAZ180 v2) | feilipu

Mark Sherman says:
on April 8, 2017 at 09:17
I got to here from your retrochallenge entry. I am completely shocked a 40 year old 2 mhz float processor still beats out a 36 mhz Z180. Now I want to figure out how many clocks an arduino takes for float division.

Reply

A WordPress.com Website.