Most of the program took inspiration from the provided C program, but with further improvements to inefficiency. There was a flaw in the C program whereby it also took the (N+1)th values, when it need not, so we updated that as well as made sure the asm implementation was correct.

The first step is how we allocate the memory for x\_store and y\_store.

x\_store

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pointer to |  | … | … |  |

y\_store

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pointer to |  | … | … |  |

.lcomm x\_store, (N\_MAX+1)\*4

.lcomm y\_store, (N\_MAX+1)\*4  
.lcomm n\_calls, 4

As can be seen above, N\_MAX + 1 words of memory are given for both x\_store, y\_store arrays. This is so we can access the number of calls, as well as the values quickly. Another word of data is used for storing the number of calls so far. The main difference between this more efficient implementation is that there is no longer an O(n) operation to make sure the values can be accessed easily. In C, this is done by moving N numbers, which is inefficient. We simply have pointers as seen above.

@Register map

@R0 - N, returns y

@R1 - b

@R2 - a

@R3 - x\_n

@R4 - x\_store addresses, n\_calls address

@R5 - y\_store addresses

@R6 - a, b values

@R7 - x\_store, y\_store values

@R8 - N counter

@R9 - y\_n

@R10 - no. of call, pointer to x\_(n-1), N\_MAX

Next is the usage of each register, which are as labelled above. Of special note is R9, which is used to calculate throughout, as well as R4, R5, which are pointers to values of x\_store and y\_store are being looked at currently.

@ b[0]

LDR R6, [R1], #4 @ Memory: 0x04916004

@ b[0] \* x\_n

MUL R9, R3, R6 @ DP: 0x00009613

@ load current no. of call n

LDR R4, =n\_calls

LDR R10, [R4] @ Memory: 0x0494A000

@ if N is greater than number of previous calls

@ update N so loop doesn't get invalid values

CMP R0, R10 @ DP: 0x0150000A

IT GT

MOVGT R0, R10

MOV R8, R0 @ DP: 0x01A08000

@ increment the no. of calls

ADD R10, #1 @ DP: 0x028AA001

STR R10, [R4]

The first step simply involve loading , and calculating , storing in R9. Since we do not want to calculate using values not available, the next part ensures we take for use in calculating using previous values. R8 is used as the counter for the loop in the next section. Afterwards, increment number of calls and store.

@ load start address to x\_store, y\_store

LDR R4, =x\_store

LDR R5, =y\_store

@ load pointer to &x\_(n-1)

LDR R4, [R4]

@ check if address exists in x\_store yet

CMP R4, #0 @ DP: 0x03540000

BEQ zero @ Branch: 0x08800054

@ &x\_(n-1) to compare later

MOV R10, R4 @ DP: 0x01A0A004

@ load &y\_(n-1)

LDR R5, [R5] @ Memory: 0x04955000

@ if no need to enter loop

CMP R8, #0 @ DP: 0x03580000

BEQ end @ Branch: 0x08800048

The next few steps are to load the addresses to x\_store and y\_store as defined earlier, and obtain the values stored in the first first few locations.

Then, check if address exists in x\_store. Declaration of .lcomm is memory in .bss, where all values are initialized to 0. If no address, skip to zero.

Move to compare the value obtained from the tracking pointer in the loop, to know when to circle back. R4 will be modified, so we use R10 to safekeep the value. Then load address to . Decide to enter the loop below using the counter.

**loop:**

@ load x[n-i] value

LDR R7, [R4], #-4 @ Memory: 0x04147004

@ check if value loaded is &x\_n-1 -> need to circle back to end of array

CMP R7, R10 @ DP: 0x0157000A

BEQ circle @ Branch: 0x08800024

@ b[i]

LDR R6, [R1], #4 @ Memory: 0x04916004

@ y\_n += b[i] \* x[n-i]

MLA R9, R6, R7, R9 @ DP: 0x00299716

@ load y[n-i] value

LDR R7, [R5], #-4 @ Memory: 0x04157004

@ a[i]

LDR R6, [R2, #4]! @ Memory: 0x05B26004

@ y\_n -= a[i] \* y[n-i]

MLS R9, R6, R7, R9

@ decrement counter

SUBS R8, #1 @ DP: 0x02588001

BGT loop @ Branch: 0xC8000028

@ get back start address of a

SUB R2, R2, R0, LSL #2

B end @ Branch: 0xE8800028

The loop is mostly self-explanatory. At the beginning is a check as to whether the tracking pointer has reached the start of the array (index 1 for x\_store, index 0 for y\_store). If so, need to circle back to get to index + N\_MAX for the next number. At the end of the loop is to check whether it has been calculated enough times, finishing the loop, or to repeat the loop. Since the step to divide by was delayed to the end, there needs to be a calculation to get the address of again.

@ load end addresses of x\_store, y\_store

@ N\_MAX bytes from start addresses

**circle:**

MOV R10, #N\_MAX @ DP: 0x03A0A00A

LDR R4, =x\_store

LDR R5, =y\_store

ADD R4, R4, R10, LSL #2

ADD R5, R5, R10, LSL #2

B loop @ Branch: 0xE8000048

@ for x\_0, y\_0 no address yet in x\_store, y\_store

@ get address into x\_store for consistency

**zero:**

LDR R4, =x\_store

LDR R5, =y\_store

STR R4, [R4]

STR R5, [R5]

circle is the to load the ending addresses of x\_store and y\_store as needed by loop. zero is simply to load addresses into the the memory for x\_store and y\_store into R4, R5, so the next section can store the values properly.

**end:**

LDR R6, [R2]

@ /a[0]

SDIV R9, R6

@ get next address to overwrite

LDR R4, =x\_store

LDR R5, =y\_store

LDR R4, [R4]

LDR R5, [R5]

@ check if at end of array

MOV R10, #N\_MAX @ DP: 0x03A0A00A

@ R6 used as temp register

LDR R6, =y\_store

ADD R6, R6, R10, LSL #2

CMP R6, R5 @ DP: 0x01560005

BNE store @ Branch: 0x18800008

LDR R4, =x\_store

LDR R5, =y\_store

After calculating everything except dividing by , we do exactly that. If the tracking pointers are at the end of the array, it’s necessary to circle back by loading the start addresses again.

**store:**

@ store x\_n, y\_n into x\_store, y\_store

STR R3, [R4, #4]! @ Memory: 0x05A43004

STR R9, [R5, #4]! @ Memory: 0x05A59004

@ store pointer to (n-1)th address for next n

LDR R6, =x\_store

STR R4, [R6]

LDR R6, =y\_store

STR R5, [R6]

@ /100

MOV R8, #100 @ DP: 0x03A08064

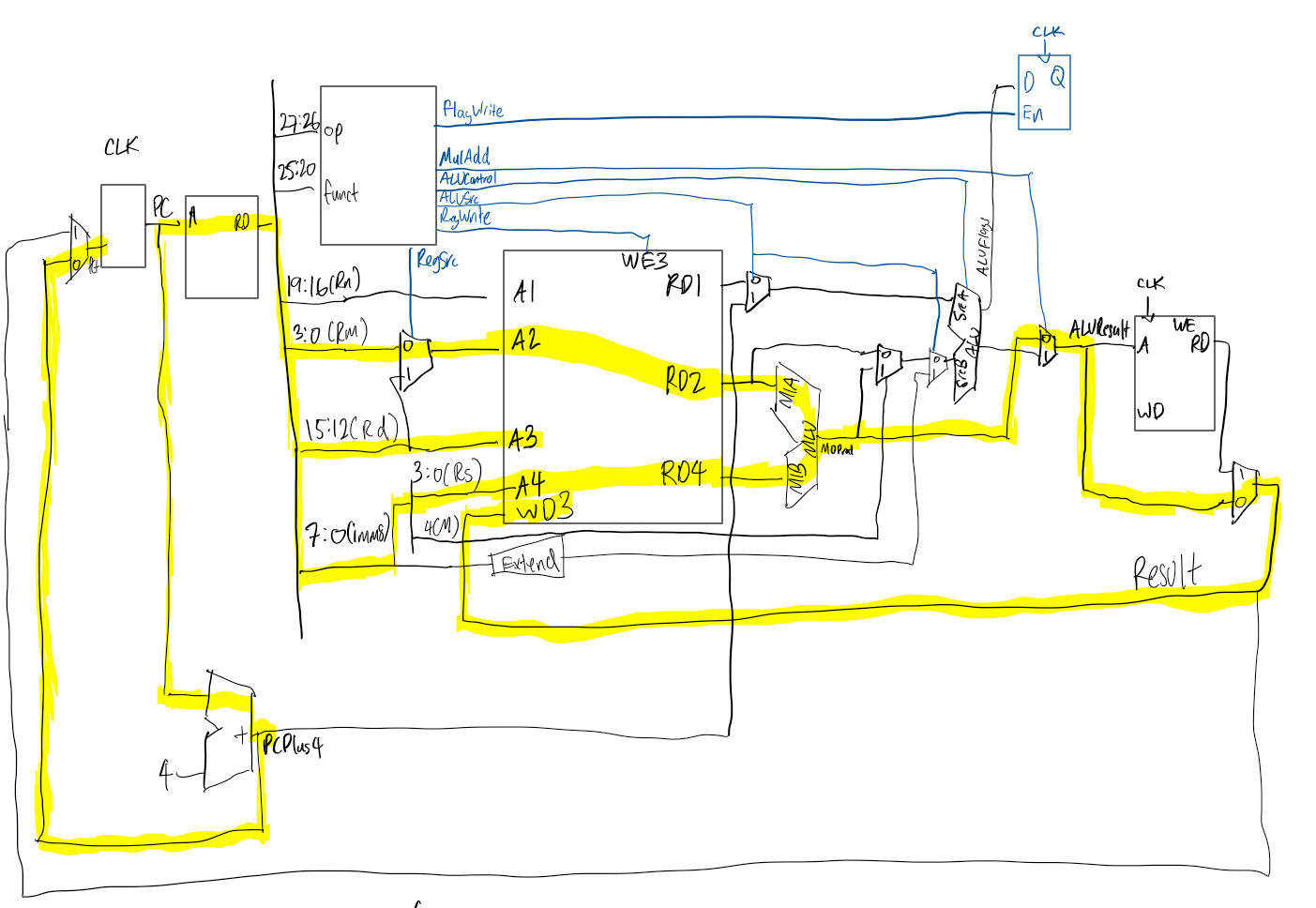
SDIV R9, R8

MOV R0, R9 @ DP: 0x01A00009

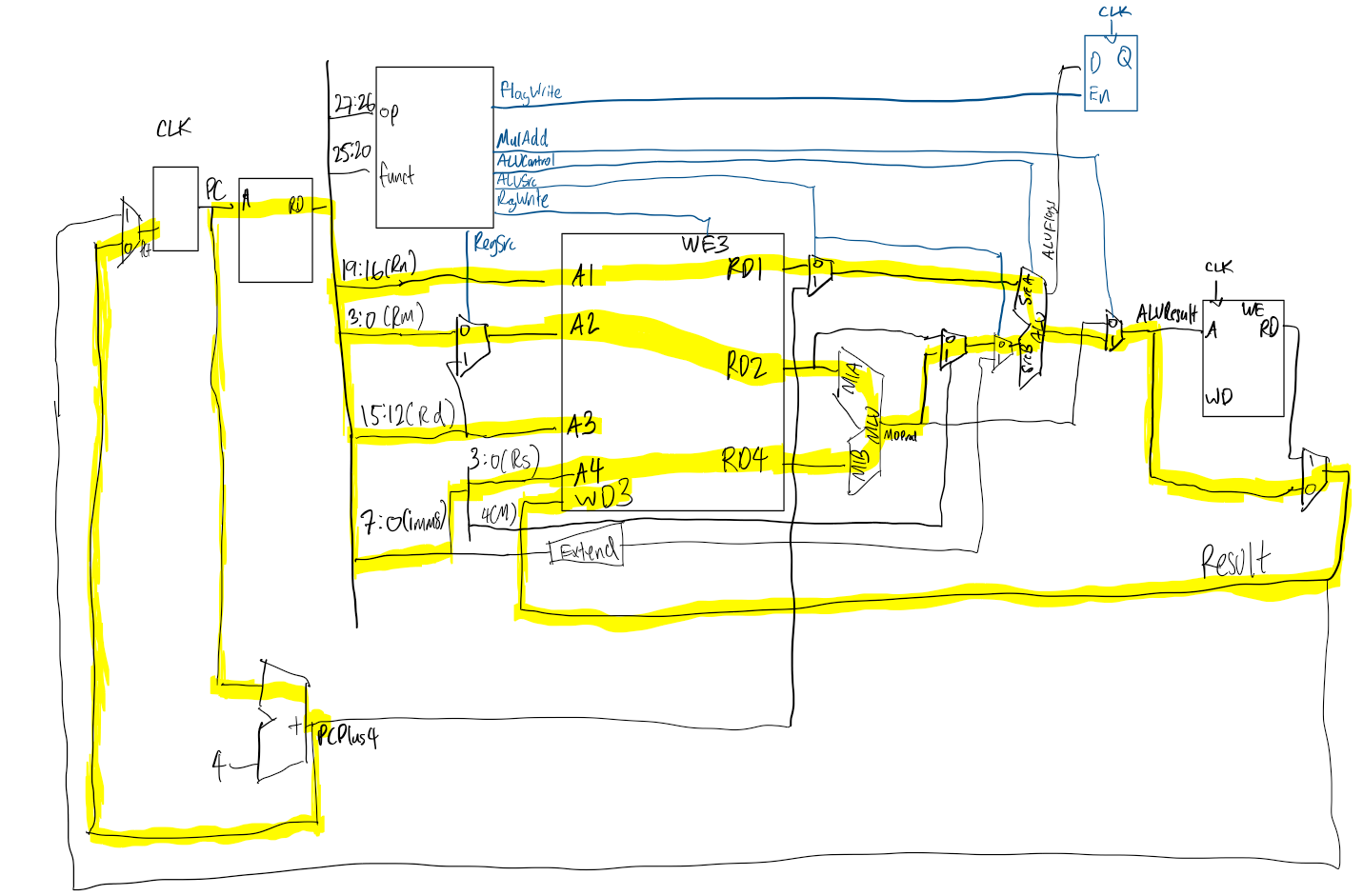
Finish the program by storing the required values into the appropriate addresses.

Datapath Architecture

MUL operation Datapath



MLA Datapath



MLU is the hardware multiplier block, with MIA, MIB, MOProd being Mult\_In\_A, Mult\_In\_B, Mult\_Out\_Product respectively.

Only the following needed for Control Unit Design

Change for ALUControl, to add whenever I==0 && M==1. I==0 is important to distinguish between bit 4 being an M bit or part of imm8 for normal DP operations.

ALUControl = (op==00)?((I==0 && M==1)?0100:cmd):(U?0100:0010)

MulAdd is simply whether to take the result from the hardware multiplier block directly (MUL), or with the addition of Rn (MLA).

MulAdd = cmd[0]