DECA Lab Spring Term

Part 2: EEP1 Control Path and Jump Instructions

Department of Electrical and Electronic Engineering

Imperial College London

v1.4

Spring 2024

Contents

1 Introduction

Before the lab

- Download from the lab1 repo eep1lab2 - its version number should be v1.9. Version numbers can be checked from properties on the top design sheet description. (This is the same as the eep1lab1 you used for lab1 - assuming you picked up the later v1.9 version).
- Read Spring Lecture 4.

Figure 2 shows all of the EEP1 jump instructions, with their machine code encoding. A jump instruction directly modifies the program counter to execute a different part of the program, either unconditionally, or conditional on some boolean condition on the EEP1 flags. In Section 2 you will focus on how the EEP1 control path in Figure 3 implements these instructions. In section 3 you will add hardware to the Lab2 design to implement the required jump conditions. Finally in sections 4 and **??** you will use jump instructions to implement multiply routines.

2 Control Path Operation

Look at the schematic of the `controlpath` block in Figure 3 and note that this block contains the EEP1 Program Counter (PC). As you can see from this sheet, PC.Q is output to the instruction memory address MEMADDR and determines the memory location of the current instruction word (MEMDATA=INS(15:0)). PC.D therefore must be the address of the **next** instruction and it is output from NEXT.PCNEXT. The NEXT block therefore determines if and where the EEP1 will jump.

The control path contains 4 1-bit flag flip-flops: FFN, FFZ, FFC, FFV. The Q outputs of these are input to the NEXT block as the 4 Flags: FlagN, FlagZ, FlagC, FlagV. In addition FlagC is output to the datapath so it can be used as an adder input in ALU operations. The D inputs of these flip-flops come from the datapath and are

set according to the flag conditions. The Q outputs of these flipflops are the *flag bits* N, Z, C, V: boolean values used to determine whether a jump will occur, e.g. a JEQ instruction will only jump if Z=1.

The flag flipflops introduce (between D and Q) a one cycle delay which allows information about data in one instruction to be used in a conditional jump in the next instruction. The `controldecode` block has outputs NZEN and CVEN that determine, based on the current instruction, whether FlagN and FlagZ, or FlagC and FlagV, should be overwritten. The ALU instruction specification states which instructions write which flags: for example, MOV will write only FlagN and FlagZ, whereas ADD will write all of the flags.

From this overview you can see that the operation of the control path depends on NEXT; this has inputs from the flags and (via `controldecode`) the instruction word. It has just one output: PCNEXT.

2.1 `Next` block

Figure 4 shows the `next` sheet. The logic that controls PCNEXT contains MUX1 and MUX2 controlled by COND.RET, COND.JUMP, and NEXT.JMP. Before doing Task 1, draw up by hand from the given hardware a truth table for how these 3 signals, and algebraic inputs NEXT.pC, NEXT.OFFSET, and NEXT.RA, determine the value of PCNEXT.

□ **Task 1.** *Create an 8-row Issie algebraic truth table, selecting components as shown in Figure 5, to examine how the NEXT sheet drives NEXT.PCNEXT. Compare the truth table with your hand-generated truth table - they should be the same. You may assume that NEXT.JMP is 1 only for jump instructions. What is PCNEXT when NEXT.JMP = 0? Compare the truth-table with the EEP1 jump instructions (Figure 2) to determine how inputs* `COND.JUMP` *and* `COND.RET` *affect the control path operation.*

3 Jump Conditions

Open the `cond` sheet from your Lab1 design in Issie. Note that JMPCOND(3:0) is equal to JMPOPC(3:0) from the current instruction word. Check that the conditions for JMPOPC(3:1)=0,1,2,3,7 are the same as those required by Figure 2 in logic already on the `cond` sheet. You may conveniently do this using an Issie Truth-table selecting MUX1 and all of the logic connected to its inputs, and then setting the Flag inputs to be algebraic.

□ **Task 2.** *In the* `cond` *sheet replace the existing constant '1' connection at MUX1 inputs by appropriate logic to implement the jump conditions for JMPOPC = 4,5,6. Show that your logic is correct by using an Issie algebraic truth-table.*

4 Multiply Software

The EEP1 does not have a multiply instruction, and therefore multiply operations must be implemented in software using combinations of shift and add instructions, with the correct jumps to implement control flow. Lecture 3 provides context, showing how shifts and addition can implement multiplication.

Figure 1 shows an algorithm for multiplication written in C++.

```
0  // implement sum := op1 * op2 (LS 16 bits of)
1  // assume int = 16 bits (16 bit version of C).
2  unsigned int op1, op2, op2_shifted, sum; // variables
3  sum = 0; // initisalise
4  op2_shifted = op2; //initialise
5  while (op1 != 0) {
6      if (op1 & 1) { // & bitwise AND operation
7          sum = sum + op2_shifted;
8      }
9      op2shifted = op2shifted << 1; // left shift by 1
10     op1 = op1 >> 1; // right shift by 1.
11 }
```

Figure 1: While loop implementation of multiply

□ **Task 3.** *Walk through the code in Figure 1 by hand with op1=12, op2 = 5 to check that it works correctly and generates 60 as an answer. Compare its operation with the shift and add multiply method explained in Lecture 3 to show how it works.*

□ **Task 4.** *Using the method from Lecture 4 for* `if-then` *and* `while` *constructs, and the EEP1 instructions for addition, shifting, logical AND, translate the program in Figure 1 into assembly language, using registers R0, R1, R2, R3 for the 4 required variables. Write your program in a* `mul1.txt` *file and use eepassembler to turn this into a* `mul1.ram` *file containing machine code. Run the file using your own EEP1 design now fully working and check that it multiplies.*

□ **Task 5.** *Write a similar multiply program so that* `op2shifted` *and* `sum` *are both 32 bits, made up of each of two registers. Replace the 16 bit add and shift operations by a corresponding 32 bit add and shift.*

5 Challenge

The challenge here is to implement additional instruction(s) that can speed up the software implementing $16\times16 \rightarrow 32$ multiplication. The challenge may be competed individually or in lab pairs. You may find that cooperating in lab pairs, with different students doing different parts, is effective. The challenge has many parts: some credit will be given for partial completion.

The points below are meant to be a guide to what you might do. You can deviate from them, except for the restrictions in hardware you can use.

1. Add to the ALU block to make an unsigned multiply instruction MULU8: `Ra := Ra(7:0)*Rb(7:0)`. This can be done in a hardware $8\times8 \rightarrow 16$ multiply block using 7 adders following the 3 bit adder example in the notes. You can code your multiply instruction by using MOVC1 as a mnemonic in the assembler as described in section below. Note that although the correct mnemonic is MULU8 the assembler will recognise instead MOVC1. The `c` field of the instruction word is input into the ALU in the SHIFTOPC and SCNT fields (note instruction machine code reference to see how it is aligned with these fields). Therefore additional logic in the ALU can distinguish between MOVC1 & MOV instructions.

2. Work out a minimum sequence of instructions to use such an unsigned multiply instruction multiple times and perform $16 \times 16 \rightarrow 32$ unsigned multiplication where the input comes from two registers and the 32 bit output is written to two different registers. It will help you to note that if a, b, c, d are all 8 bit unsigned numbers:
$$(a + 2^8b) * (c + 2^8d) = a * c + 2^8(a * d + b * c) + 2^16b * d$$

3. Record the speed of your instructions (in cycles executed per $16 \times 16 \rightarrow 32$ multiply) using the new MUL8 instruction and compare it with the speed of a software multiply without this instruction

4. Suppose the main disadvantage of extra hardware is that additional delay through ALU combinational logic slows down the CPU clock. Assume that the delay through any path from input to output of adders is the only significant combinational logic delay. Determine the maximum delay of any path (as a number of adders) from Ra or Rb to the register file input in your design. Thus, for example, the ADD instruction has one adder delay from inputs to outputs: and some extra hardware which we ignore.

5. Work out additional instructions (using as required MOVC2, MOVC3, MOVC4 and choosing your own correct mnemonics) that use the same or a similar multiply unit but reduce the number of instructions required to perform a complete $16 \times 16 \rightarrow 32$ operation. You are not allowed to use more than 8 adders but they can each have lengths of up to 24 bits. You are not allowed to use ROMs. There is no limit on the other hardware (MUXes etc) that you use. Document precisely the function of your instructions and the new code that performs multiply.

6. How can signed $16\times16 \rightarrow 32$ be implemented using your unsigned multiply instructions? Would an additional instruction help this be more efficient? Can you devise such an instruction reusing the same adders?

5.1 Design Notes

All of the points below will help you if you are able to implement them:

1. Use hierarchy to simplify your hardware design (this will also speed up design and testing!). As an example look at the `shift` block you are given as part of EEP1. Be creative: you do not have exactly to follow examples in the notes. Try to re-use hardware (for example adders) when two different and related operations need to be implemented.

2. Try to work out optimal sequences of instructions. Consider whether slightly different hardware could speed up the software.

3. Analyse the performance of your design: both number of instructions and overall number-of-adders delay.

4. Be alert to the possibility that your design works only partially. Find good ways to test it and document those.

5.2 Implementing Unused Instructions

Unused machine codes are also available for `INS(15:13=0b111)`. These could be used to implement two more ALU instructions, modifying the datapath decode logic.

The 7 *extra* MOV instructions, which correspond to register-operand MOV instructions with the c register field non-zero, could be used. The assembler recognises `MOVCn Ra Rb` where $n = 1..7$ and generates the correct machine code. You may use these instructions with $n = 1, 2, 3, 4$ to interface with additional hardware. For example `MOVC1 Ra, Rb` could be used to implement Ra := Ra op Rb, where `op` is some new operation you have defined in the ALU. You may not use $n = 5, 6, 7$, these are reserved for other usage (later).

The `MOVCn` instructions can be used in the context of a co-processor to read or write multiple additional registers, with a (for write) or b (for read) fields selecting one of up to 8 registers.

5.3 Collaboration: IEEE 754 half-precision (binary16) Floating Point

For additional credit, optionally, combine hardware and software to make a fast IEEE-754 binary16 multiply routine. This is quite challenging. To simplify the task you may assume that *subnormal* and *special* (infinite or NaN) numbers do not occur as either inputs or outputs. The standard normalised sign-magnitude-exponent coding described in the reference is thus always used together with, as a single special case, zero. IEEE-754 specifies detailed rounding requirements. For this challenge you need not implement correct rounding as long as the result is one of the two closest possible floating points numbers to the exact value.

For your interest, a complete floating point library would use this multiply routine, and similar routines for addition and subtraction. Division (or rather reciprocal from which division can be calculated) can be implemented using multiplication and a Newton-Raphson algorithm. Such a complete library can therefore be implemented quite efficiently in software given only a fast multiply algorithm.

Challenge

Implement faster EEP1 $16 \times 16 \rightarrow 32$ multiplier using hardware with no more than 8 adders (each up to 24 bits in length) and one or more instructions that use this hardware.
If you have time, tackle the IEEE-754 challenge.

EEP1 Jump Instructions

EEP1 machine code	15	14	13	12	11	10	9	8	7	6	5
	1	1	0	0	JMPOPC						

MPOPC(3:1)	NZCV condition	Condition on data for jump with JMPOPC(0)=0	JMPOPC(0) = 0	
0	1	always	**JMP** (Always jump)	**NO**
1	Z	result = 0	**JEQ** (Equal)	**JN**
2	C	C=1 ≡ u(Ra) ≥ u(Op)[1]	**JCS** (Carry Set / Unsigned Higher or Equal)	**JC**
3	N	z(result) < 0	**JMI** (Minus)	**JP**
4	$\bar{N} \oplus V$	z(Ra) ≥ z(Op)[1]	**JGE** (Signed Greater or Equal)	**JL**
5	$(\bar{N} \oplus V).\bar{Z}$	z(Ra) > z(Op)[1]	**JGT** (Signed Greater Than)	**JL**
6	$C.\bar{Z}$	u(Ra) > u(Op)[1]	**JHI** (Unsigned Higher)	**JL**
7	1	always	***JSR*** *(R7[4] := PC + 1, PC := PC + JOFFSET)*	***RE***

JGE,JGT,JHI, JCS jump conditions jump based on the flag values. The table column 3 shows the comparison of **Ra**
`a, Rb` instruction which delivers those NZCV values, or the condition on an ALU result which delivers the requi

JSR and **RET** always jump and are used to *jump to* and *return from* subroutine see notes. **RET** does not use **Joffse**

JMPOPC(0) = 1 inverts all conditions, except for JSR/RET

Hardware is simpler because **JSR**/**RET** write/read R7 since **Ra** is normally read and written and for these instructi

ump Instructions	Assembly	Operation	Examples
ll except RET, JSR	JEQ JOFFSET; JEQ SYMBOL	PC := PC + JOFFSET if condition is true	**JMP −1**
ET	RET	PC := R7	**RET**
SR	JSR JOFFSET	R7 := PC+1; PC := PC + JOFFSET	**JSR 10**

Figure 2: EEP1 Jump Instructions

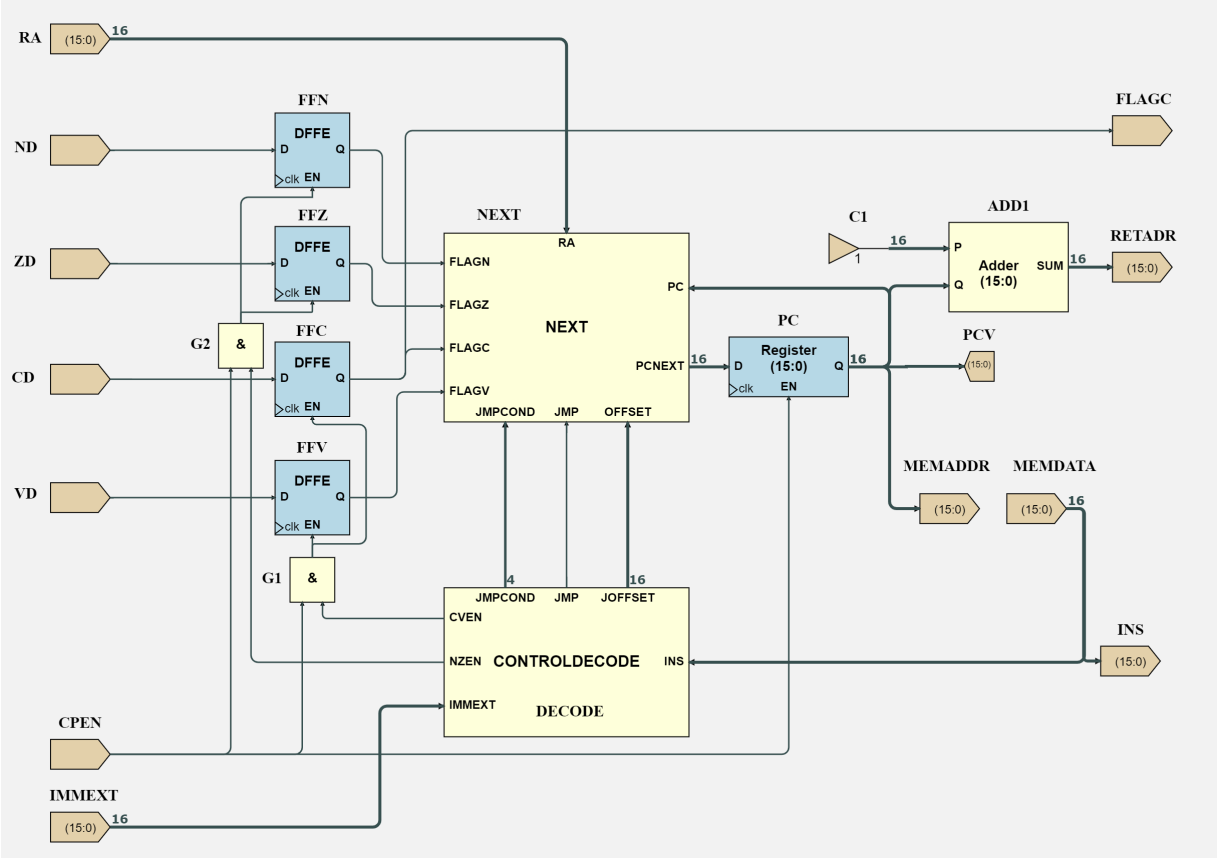

Figure 3: EEP1 Control Path

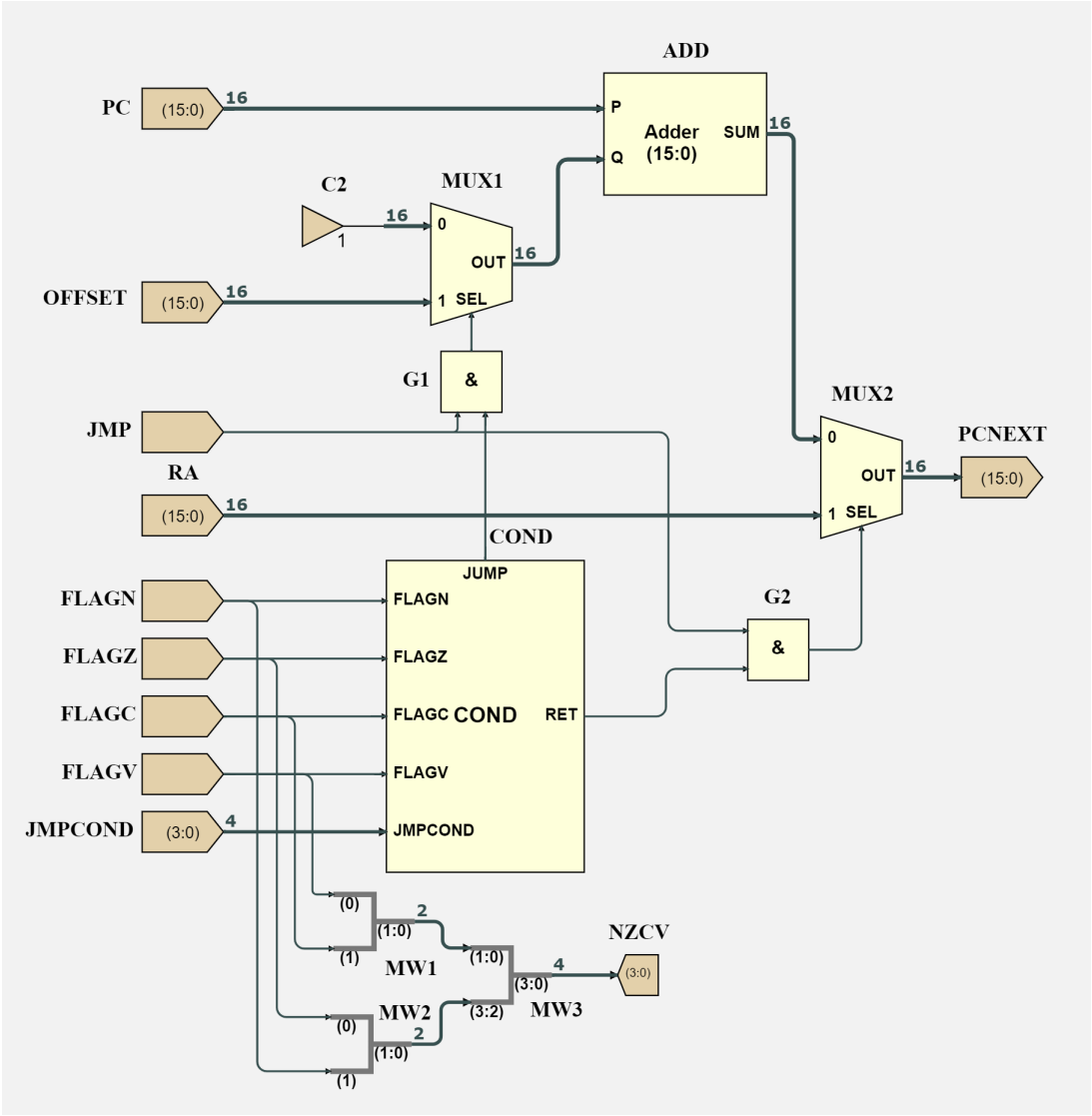

Figure 4: EEP1 Next Block

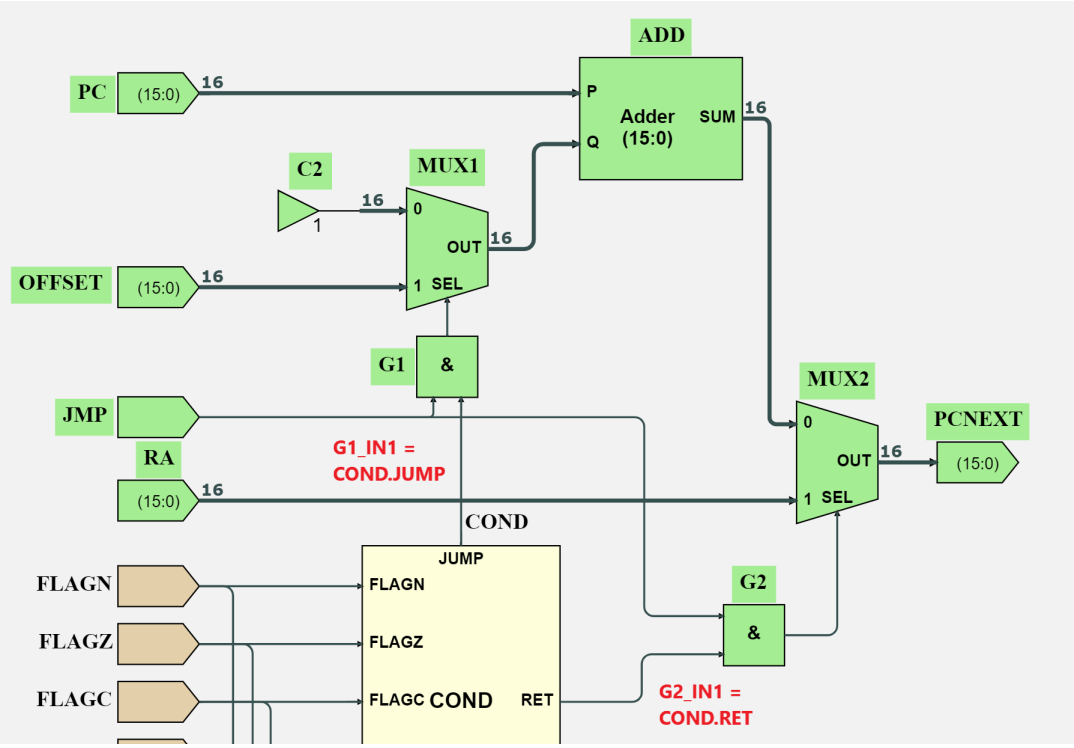

Figure 5: Component Selection to generate PCNEXT truth table: red text indicates resulting truth table column headers