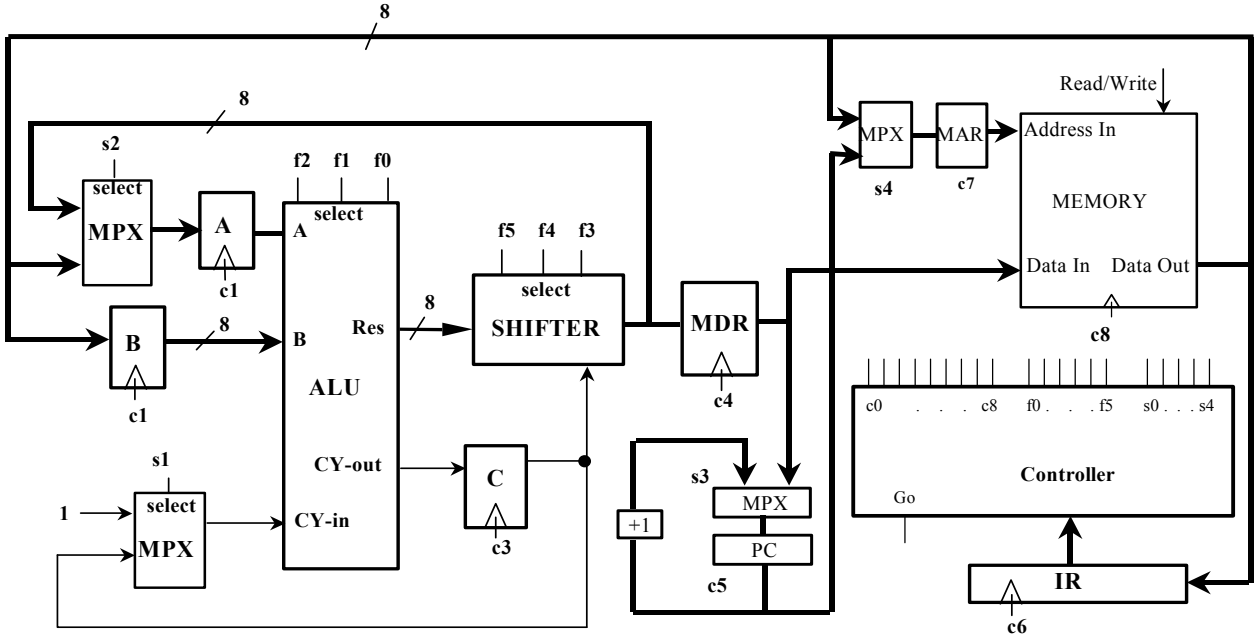
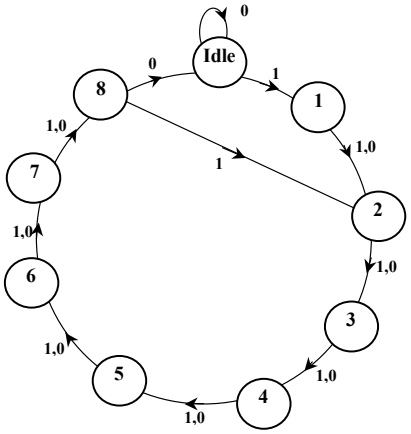
Lecture 17: Designing a Central Processor Unit

With the end of term approaching Henry and Duncan were having a chat (yes they do occasionally talk to each other) about the hardware course, and how Christmas was coming and how nice IC would be if there weren't any students, when suddenly they had a brilliant idea. Henry had designed a manual processor, and Duncan had designed a random access memory - why not put them together and build a processor. "Aha", they thought, "we can then consult the local software guru (Tony) on how to make a Haskell Interpreter and then see if we can sell the idea to Bill Gates". How disappointed they were when they found out that the Intel corporation had got there first. Anyway, here was the first attempt at putting the two together.



The manual processor put its result in a result register, but now the result could be put directly into the memory, so the MDR replaces the old RES register. A big problem was where in the memory to save the results of a computation. The first attempt at a solution was to change the former four byte instructions into five byte instructions, with the last byte storing an address for the result. So our crack design team provided a direct path from the memory to the MAR. They also tidied up the memory so that the data out line was separated from the data in, and always enabled. Lastly, instead of using the bits of the IR directly to set the function of the ALU and shifter and multiplexers, they are used as inputs to the controller to provide more flexibility. All that remained was to design a controller.



State	Function
Idle	Wait for the start signal
1	Set address to get next byte
2	Load the instruction, Set address to get next byte
3	Load Data to Reg A Set address to get next byte
4	Load Data to Regs B and C Set address to get next byte
5	Load the Instruction
6	Load Data to Regs C and MDR Set address to get next byte
7	Load Result Address to MAR
8	Store the Result Set Address to get next byte

The controller is a finite state machine, and can be designed using our synchronous design methodology. The plan for the controller would be to combine the manual processor controller (which just cycled through five states, assuming the correct data on the data in line) with the fetch cycle for the memory. Thus every time we need a new byte we do a fetch. A basic outline controller is shown. The only input that is shown is the one marked "go" in the processor design, which essentially starts and stops the processor. But there are the other inputs from the IR, and the outputs will depend on these so we will be designing a Mealey machine in this case. The output logic of the controller is to produce the outputs that do three things:

1. Set the multiplexers so that the data goes to the right places.
2. Set the functions of the ALU and shifter so that the correct calculations are done

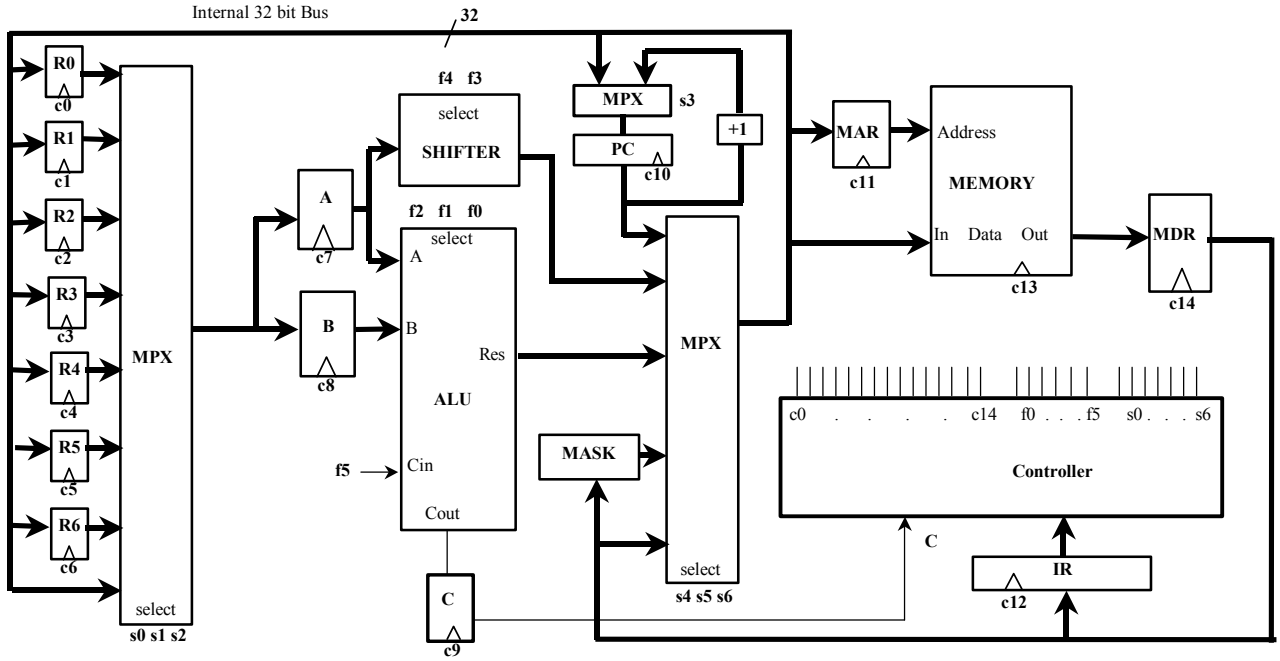
3. Enable the clocks of the registers that are to receive data

Duncan and Henry designed a controller for a few instructions (ADD, NEGATE, JUMP and so on), and then took their design to the software department. The software manager (Susan) looked at it in horror. "Are you guys living in the stone age?" she asked. "This machine will be slow!" We pointed out that it was for a Haskel interpreter, but it was no use, we had to think of ways of speeding it up. Here are some ideas for doing this.

1. Scale up the architecture: instead of doing everything in 8-bit blocks, why not use 32 bit blocks. Everything in the design stays the same but uses four times as many gates. However, immediately we get a speed up because we write or read four times as much information to and from the memory in the same clock times. If we do arithmetic we can do it at full precision with no need for carries.

2. Most of our computations will require local storage, so rather than saving everything in memory, it would be much faster if we provide a number of central processor registers. We opted for seven registers in this design, but we could have used many more if the hardware costs could be met.

Thus our final design looked like this:



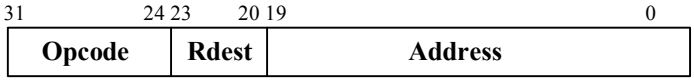
One choice may seem rather odd. Why did we retain the registers A and B. There were two reasons. As hardware designers we need registers that we can manipulate, but the programmers don't know about. Hence we can change their contents without crashing a program. We will need them to build the correct sequence of operations to implement instructions. Secondly, without them we have a long chain of combinatorial circuits (MUX to ALU to MUX to Internal Bus), and this may cause us problems with spikes when we try to make the clock go as fast as possible. In order to increase the speed of the combinational logic, we placed the shifter and the ALU in parallel, and opted for a simple four function shifter. The four functions will be shift right arithmetic (duplicating the top bit), shift right logical (setting the top bit to zero) and shift left (setting the bottom bit to zero) and no change. Noting that the memory will not have the power to drive lots of different registers and multiplexers, we connected it only to the MDR. We also introduced a combinatorial circuit called "MASK" in the MDR line. We will discuss its function in due course. Lastly, we noted that since we now have a 32 bit ALU we will be able to do all our arithmetic operations on single words, and so we do not need elaborate carry in carry out arrangements. Thus, the carry in to the ALU can be set to zero or one only, and the carry out is used only to indicate if arithmetic overflow has occurred. We therefore decided to scrap the multiplexer selecting the carry in and make it a single function line (f5).

The next problem is designing the controller. The function of the controller is to provide the correct operation sequences to carry out the program instructions. Thus it was necessary to determine the instruction set, and so back we went to the software department.

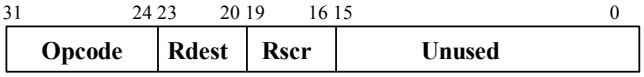
The first decision that we reached is that we will have less than 255 different instructions, and the top eight bits of the instruction word will determine the instruction. It will be called the "Opcode". Next we decided that the data carried in the instructions (if any) would be stored in bits 0-19. This enabled us to separate out any data from the opcode with a very simple MASK circuit. The MASK connects bits 0-19 directly from input to output, and connects outputs 20-31 to ground. Thus it does not contain any gates at all! Any unsigned integer on bits 0-19 of the input, becomes the same, unsigned integer in 32 bits on the output. While this choice does

keep things simple, it restricts the data stored in an instruction to 4 Mbyte (since we are going to fetch 4 byte words from memory). We are now in a position to define our instruction set.

Memory Reference Instructions.
We decided to get by with just four memory reference instructions. These are LOAD, STORE, JUMP and CALL. All other instructions will just do things with the processor registers With the exception of jump, these each require a register to be specified. The format of the instruction will be:

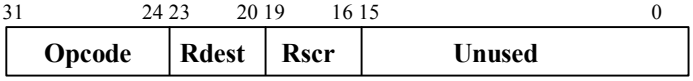


Of course the software department didn't like this. They wanted to use as much RAM as they could lay their hands on, not just the 1Mword provided by 20 bits. To keep them happy we suggested using indirect memory reference instructions. These used one of the registers as a pointer to memory, with the following format:

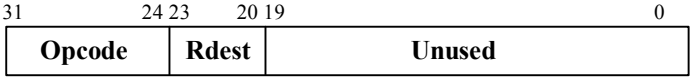


The meaning of, for example, LOADINDIRECT is load Rdest from the memory location whose address is stored in Rscr. This would allow the software department to use 2^{32} words of RAM, so they were happy with that.

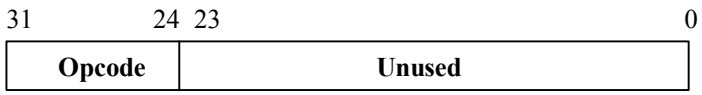
Two register instructions
These operations are internal to the processor. We will use: MOVE, ADD, SUBTRACT, COMPARE, AND, OR and XOR



One register instructions.
These are also all internal instruction that we want to carry out on a single register. They are: CLEAR, INCREMENT, DECREMENT, COMPLEMENT, ASL (Arithmetic Shift left), ASR(Arithmetic Shift Right) LSR (Logical Shift Right) and RETURN (from a subroutine). They will all have this format:



No register instructions
SKIP instructions will just skip the next instruction in the program, which will probably be a jump instruction. They normally work on the result of an arithmetic operation or a compare, and we will incorporate three possibilities SKIP, SKIPPOSITIVE, SKIPNEGATIVE. The software department were not pleased with this. The wanted a skip if result equals ZERO. However, we pointed out that to do this in hardware seriously complicated the design, and they could get round it in software. We only had the capability of storing the carry of an arithmetic result, and this was all we could use. Thinking about it, we could have been more helpful by providing a 32 input OR gate to test if all the result bits of the ALU were zero. The result of this could be kept in another 1 bit register. However, we wanted to keep the design simple at all costs. The only other no register instruction that we needed as the "No Operation" instruction (NOP) which just does nothing.



Input and Output
You may be wondering about how we get programs and data into the memory and get results out. This puzzled us at first, however, we found out that somewhere in the snow capped heights of Huxley building (around the 5th floor) there lived a mystic priest (called Naranker) who made strange and wonderful pronouncements that

nobody really understood, about a subject called architecture. He told us that all we needed to do was provide some connectors to the address and data bus (and perhaps a few lines of the controller) and he would supply us with input and output devices that we could read just as if they were memory.

Register Transfers

Having established which instructions we were going to implement, we gave the software department due warning that from now on they couldn't change their minds, since we have now to commit these instructions to hardware, and making changes won't be easy. The first stage of the process is to formalise a specification of each instruction, and to do this we determine the register transfers that will be required to execute each of our instructions. This process will clarify whether the hardware design is sufficient, and may suggest to us some possible improvements. Again we will consider these in the four groups, starting with the memory reference instructions. We will name the processor execute cycles E1, E2, E2 and E4, and we will design a finite state machine to execute them in the next lecture.

LOAD Rdest, Address	E1	MAR←MDR	Use the mask
	E2	MDR←Memory	
	E3	Rdest←MDR	No Mask
STORE Rdest, Address	E1	MAR←MDR, A←Rdest	Use the mask
	E2	Memory←A	via the Shifter (no change)
JUMP Address	E1	PC←MDR	Use the mask
CALL Rdest, Address	E1	PC←PC+1	
	E2	Rdest←PC	
	E3	PC←MDR	Use the mask

LOADINDIRCET Rscr,Rdest	E1	A←Rsrc	
	E2	MAR←A	
	E3	MDR←Memory	
	E4	Rdest←MDR	No Mask
STOREINDIRECT Rscr,Rdest	E1	A←Rscr	
	E2	MAR←A;A←Rdest	via the shifter (no change)
	E3	Memory←A	via the shifter (no change)
JUMPINDIRECT Rscr	E1	A←Rscr	
	E2	PC←A	via the shifter (no change)
CALLINDIRECT Rscr, Rdest	E1	PC←PC+1;A←Rscr	
	E2	Rdest←PC;	
	E3	PC←A	via the shifter (no change)

Although similar in nature, we see that the indirect address instructions will take longer to execute. It turns out that only LOADINDIRECT requires four execution cycles, so we wondered if we could change the hardware design a little to reduce it to three. However the marketing department told us that our mark one processor should be on the market as quickly as possible. If we could then make a mark two processor, which goes faster, we could make all our clients upgrade their machines and thus buy two processors. After all, INTEL have been doing just that for nearly thirty years.