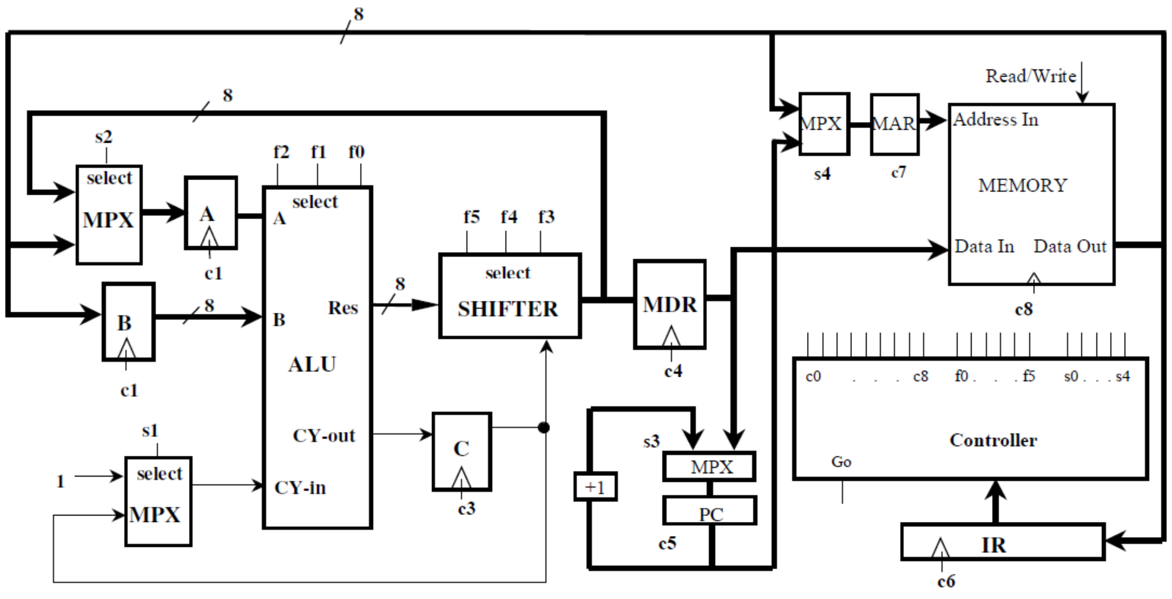
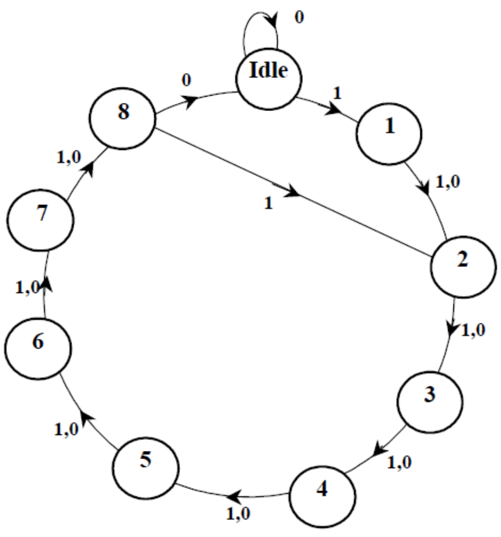
Lecture 17: Designing a Central Processor Unit 1: The Architecture

Having got to the stage where we have designed a manual processor and a random access memory, why don't we put them them together and make something useful? We can consult our local programming guru (Tony) on how to make a Haskell Interpreter and then see if we can sell the idea to Bill Gates. Don't get too excited though, because it seems that the Intel corporation may have already done something similar (about forty years ago). Anyway, here is our first attempt at putting the two together.



The manual processor put its result in a result register, but now the result must 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 use the fifth byte of the instruction to store an address for the result. For this idea we need a direct path from the memory to the MAR. To speed things up we can re-organise the memory so that the data out lines are separated from the data in, and always enabled. (Note that what we called "Data In" for the manual processor is now called "Data Out" from the memory). 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 remains is to design a controller, which is just a synchronous design problem, solvable using our synchronous design methodology.

The simplest plan for the controller is to combine the manual processor controller (which just cycled through five states, assuming the correct data was 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 looks like this:



State	Function
Idle	Wait for the start signal
1	Set address to get the next byte
2	Load the instruction Set address to get the next byte
3	Load data to register A Set address to get the next byte
4	Load data to registers B and C Set address to get the next byte
5	Load the instruction
6	Load data to registers C and MDR Set address to get the next byte
7	Load result address to MAR
8	Store the result Set address to get the next byte

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:

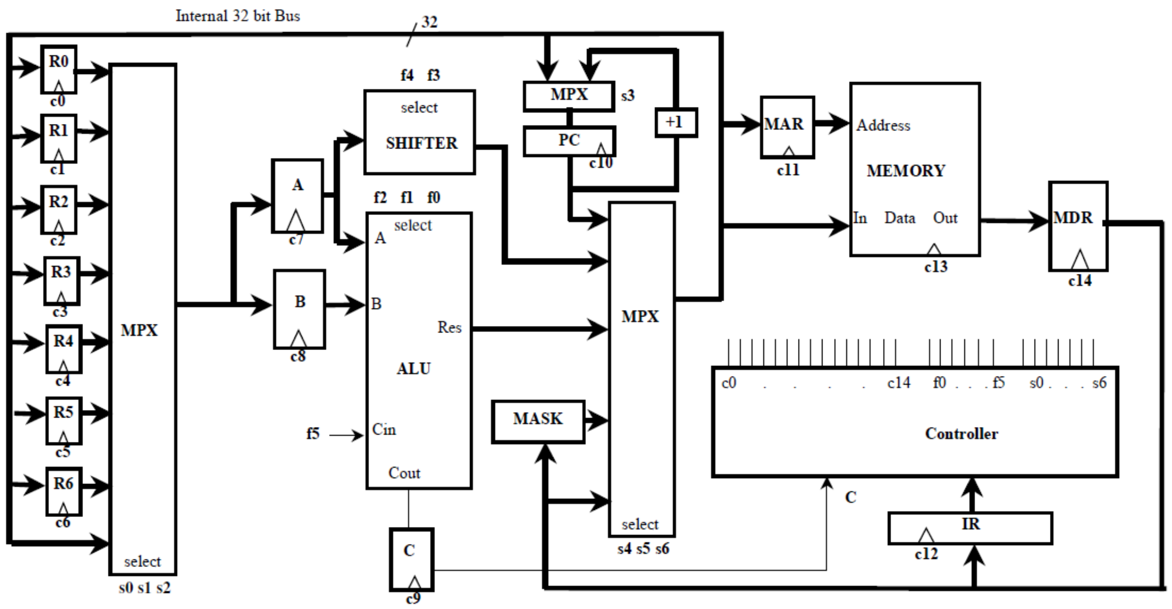
1 Set the multiplexers so that the data goes to the right places.
2 Set the functions of the ALU and shifter so that the correct calculations are done
3 Enable the clocks of the registers that are to receive data

We can now design our controller following the standard methodology that we know and love (or do I mean are sick and tired of). However I don’t think we will get very far with the venture capitalists. The resulting computer is going to be far too slow, even for a Haskell interpreter. We have 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.

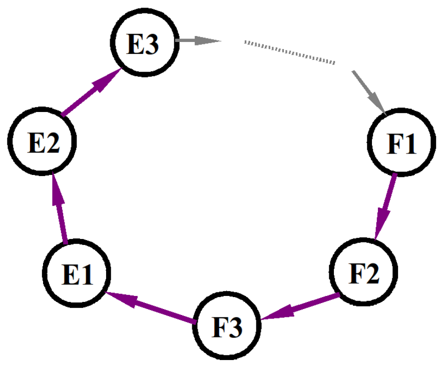
2. Most of our computations will require local storage, so rather than saving everything in memory, it would be much faster if we provide a number of central processor registers. We opted for seven registers in this design, but we could have used many more if the hardware costs could be met.

The next version of our design looks 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 dont 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 the basic four function shifter design. The four functions will be (00) hold (01) shift left with zero as carry in, (10) shift right arithmetic (duplicating the top bit), (11) rotate right (setting the top bit to bit zero). Noting that the memory will not have the power to drive lots of different registers and multiplexers, we connected it only to the MDR. Remember that large fan-outs cause time delays, so this may be the fastest option. 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) provided by the controller.

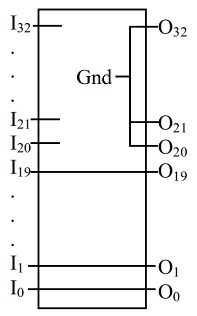
The next problem is designing the controller. We know that it will be a finite state machine that will first fetch a program instruction from the memory and then provide the correct operation sequences to execute that particular instruction. However at this stage we don't know what steps will be required, only that there will be some fetch states (F) folowed by some execute states (E).



Before we can proceed any further we must define the instruction set, and to do this we must negotiate with the software gurus.

The first decision that we reached is that we will have at most 255 different instructions, and the top eight bits of the instruction word will define the instruction. It will be called the Opcode. Since most instructions are going to use the internal processor registers we decided that the next four bits (22-20) would simply store the index of a destination register. We used four bits rather than three so that we can expand the design to have sixteen registers in the next version.

We decided that the data carried in the instructions (if any) would be stored in bits 19-0. This enables us to separate out any data from the opcode with a very simple MASK circuit shown on the data path diagram and in detail on the diagram below. The MASK connects bits 19-0 directly from input to output, and connects outputs 31-20 to bit ground. Thus it does not contain any gates at all! Any unsigned integer on bits 19-0 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 1M.

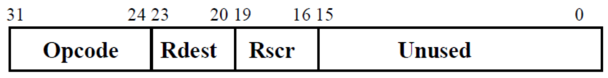


Memory Reference Instructions

The main memory reference instructions are responsible for getting data in and out of memory, and providing branching in the program instruction sequence. They will all need to refer to an address in memory. The basic ones we need are LOAD, STORE, JUMP and CALL. All other instructions will just do things to the processor registers. With the exception of JUMP, each instruction requires a register to be specified. The format of the instruction will be:

31 24	23 20	19 0
Opcode	**Rdest**	**Address**

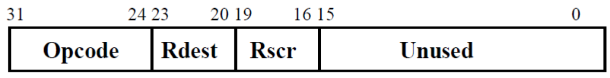
For example, the LOAD instruction will take the word stored at the address specified in the bottom 20 bits, and load it into the register specified by Rdest. Of course the software department didn't like this. They wanted to use as much RAM as they could lay their hands on, not just the 1M Word provided 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 16M words of RAM, so they were happy with that (for the time being).

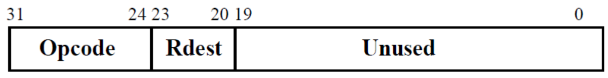
Two register instructions

These operations are internal to the processor. They involve arithmetic carried out on the internal registers. We will use: MOVE, ADD, SUBTRACT, COMPARE, AND, OR and XOR. For example ADD will replace the contents of Rdest with Rscr+Rdest.



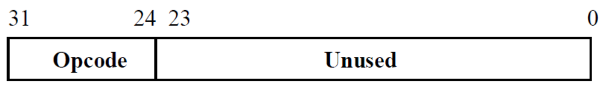
One register instructions

These are internal instruction that carry out operations on a single register. They are: CLEAR, INCREMENT, DECREMENT, COMPLEMENT, ASL (Arithmetic Shift left), ASR(Arithmetic Shift Right) ROR (Rotate 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 the 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 was the No Operation instruction (NOP) which just does nothing.



Our instruction set looks rather wasteful of memory (lots of unused bits), but we can fix that problem later.

Input and Output

You may be wondering about how we get programs and data into the memory and get results out. This looks puzzling at first, however, in term two you will be studying a strange and wonderful mystic subject that nobody really understands called "computer architecture". The architects tell us that all we need to do is provide some connectors to the address and data bus (and perhaps a few lines of the controller) and they will supply us with input and output devices that we can read just as if they were memory.

Register Transfers

Having established which instructions we are going to implement, we must give the software department due warning that from now on they can'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.

Memory reference Instructions

The memory reference instructions are defined in the following two tables.

Instruction	Cycle	Transfers	Path
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

Instruction	Cycle	Transfers	Path
LOADINDIRCET Rdest, Rscr	E1	A←Rsrc	
	E2	MAR←A	
	E3	MDR←Memory	
	E4	Rdest←MDR	No Mask
STOREINDIRECT Rdest, Rscr	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 Rdest, Rscr	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 might consider whether 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. A strategy that INTEL have been following for over thirty years.

Two Register Instructions

Instruction	Cycle	Transfers	Path
MOVE Rdest, Rsrc	E1	A←Rscr	
	E2	Rdest←Shifter	Shifter set to no change
ADD Rdest, Rsrc	E1	A←Rscr	
	E2	B←Rdest	
	E3	Rdest←ALUres;C←ALUcout	ALU=A+B, Cin=0
COMPARE Rdest, Rsrc	E1	A←Rscr	
	E2	B←Rdest	
	E3	C←ALUcout	ALU=A-B, Cin=0

SUBTRACT, AND, OR and XOR are all done the same way as ADD, they just have different ALU settings. COMPARE is just a subtract with a check to see that the result is zero. Because we have not incorporated any hardware to test to see if the ALU is all zero, all we can do is check the carry. This will be zero if Rscr≤Rdest, and 1 otherwise. Since the COMPARE result is used by the SKIP instructions, the carry register must be an input to the controller.

One Register instructions

Instruction	Cycle	Transfers	Path
CLEAR Rdest	E1	Rdest←ALUres	ALU = zero out
INC Rdest	E1	A←Rdest	
	E2	B←ALUres	ALU = zero out
	E3	Rdest←ALUres; C←ALUcout	ALU=A+B, Cin=1
DEC Rdest	E1	A←Rdest	
	E2	B←ALUres	ALU = -1 out
	E3	Rdest←ALUres; C←ALUcout	ALU=A+B, Cin=0
COMP Rdest	E1	A←Rdest	
	E2	B←ALUres	ALU = -1 out
	E3	Rdest←ALUres	ALU=A eor B
ASL Rdest	E1	A←Rdest	
	E2	Rdest←Shifter	Shifter set to Arithmetic left
RETURN Rdest	E1	A←Rdest	
	E2	PC←Shifter	Shifter set to no change

The other shifts will be done in the same way as ASL (arithmetic shift left). They have the same number of cycles, but the shifter settings will change. Notice that the COMP instruction simply flips the bits of the destination register. In order to negate a register we would need to use a COMP followed by and INC instruction.

No Register Instructions

All the skip instructions either increment the program counter or do nothing. The NOP instruction would not even require one cycle to execute.

Instruction	Cycle	Transfers	Path
SKIP	E1	PC←PC+1	

Limitations of the design

The register transfers suggest possible improvements to the hardware design. For example, if we consider the INC, DEC and COMP instructions we see that in each case register B is loaded from the main bus, where register A is loaded from the programmers registers. Perhaps another multiplexer could be incorporated to allow both these operations to take place at the same time. There are lots of other possible improvements if you look carefully, but we are in a headlong rush to get the mark one processor out on the market.