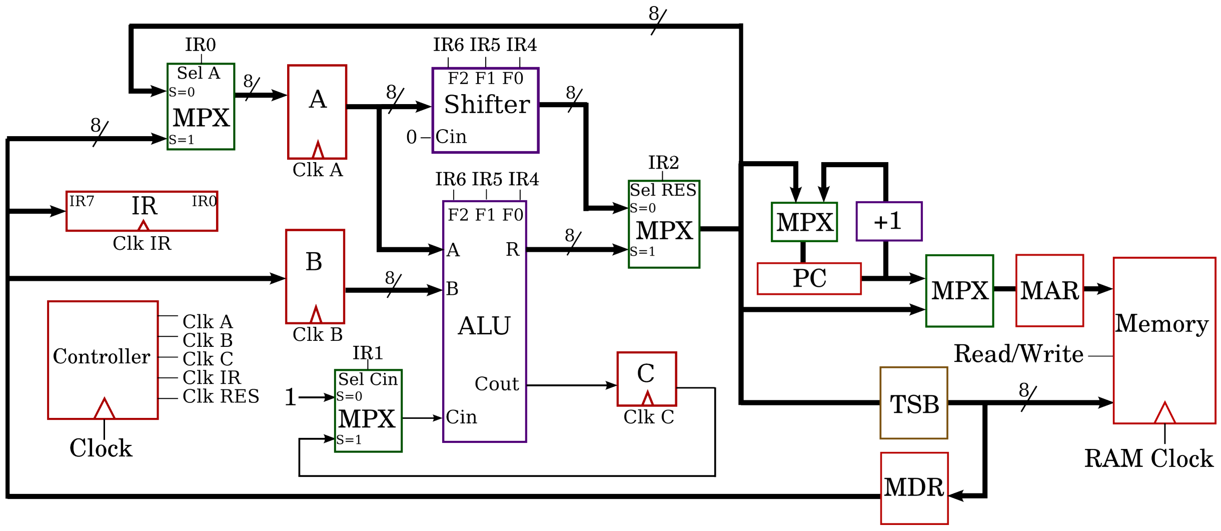
Lecture 17: Designing a Central Processor Unit 1: The Architecture

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.



Unfortunately we still have many problems to solve. We need to fetch all the data bytes from memory, which means that in addition to the five execute states of the manual procecssor we need a further two or three fetch states to execute one instruction. The manual processor put its result in a result register, but now the result must be put directly into the memory. We have to decide where in the memory to save the results of a computation, and how to get that memory address into the MAR. This could be solved by making the memory address a part of program instruction - perhaps it could go as a fifth byte to be loaded. It would need to be read directly into the MAR, so we would need a bigger multiplexer to replace Sel RES, or we would need to take it through register A. In any event we will need more steps in the execute cycle to do this. The resulting computer is going to be far too slow, even for a Haskell interpreter. I don't think we will get very far with the venture capitalists. We have to think of ways of speeding it up, and here are some things we could do.

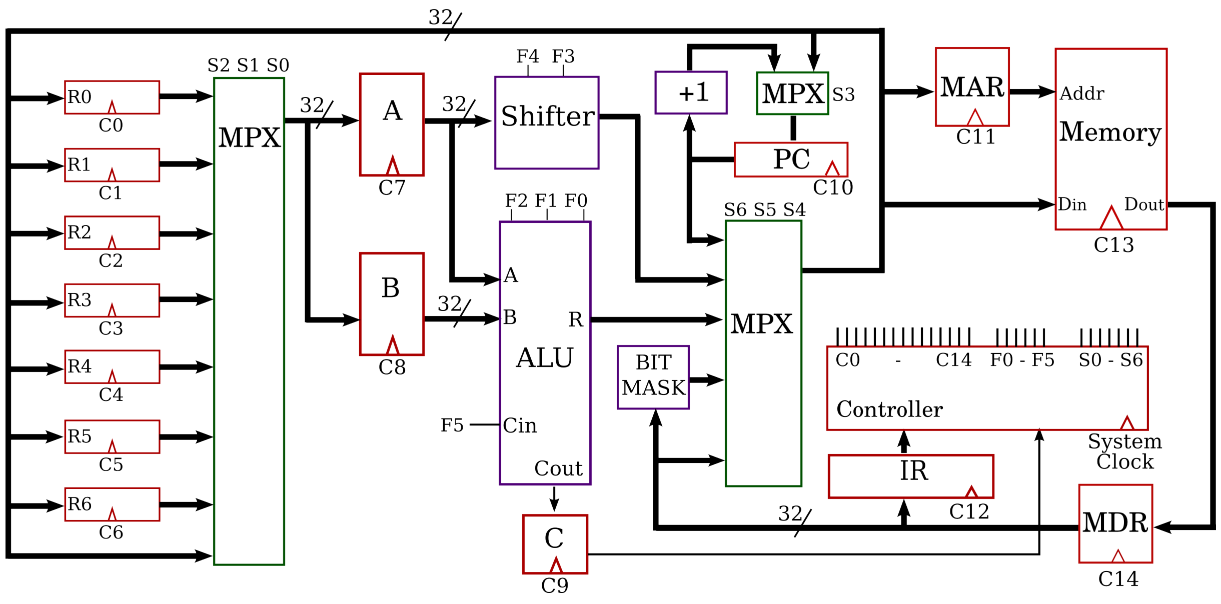
1. Scale up the architecture: instead of doing everything in 8-bit blocks, why not use 32 bit blocks (or even 128 bit blocks!). Everything in the design stays the same but uses four times as many gates. However, immediately we get a speed up because we write or read four times as much information to and from the memory in the same clock times. We can fetch all our instruction bytes at the same time. 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.

3. 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. This will allow us to design a set of really useful instructions.

4. We can re-organise the memory so that the data out lines are separated from the data in, and always enabled. The bi-directional bus saves space, but in this case optimising speed may be a better policy.

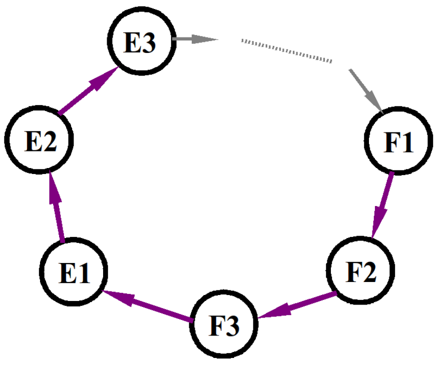
The next version of our design looks like this:



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 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 (MPX to ALU to MPX to Internal Bus), and this may cause us problems with spikes when we try to make the clock go as fast as possible. The ALU is the same design we used in the manual processor, only scaled up to 32 bits. 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 the elaborate carry in/carry out arrangement of the manual processor. The carry out is used mostly to indicate if arithmetic overflow has occurred, and therefore it is made an input to the controller. We will gain flexibility in the design of instructions if we allow the controller to determine the carry bits for the ALU. We will be able to include instructions like "branch on carry" which underpin "if .. then" program instructions. Given that with 32 bit word lengths we won't need multiprecision shifts we will have no need for the shifts with carry in and carry out. Instead we can use a 32 bit version of the four function shifter we designed previously. The functions were:

00 (A)	01 (B)	10 (C)	11 (D)
Hold	Arithmetic Shift Left	Arithmetic Shift Right	Rotate Right

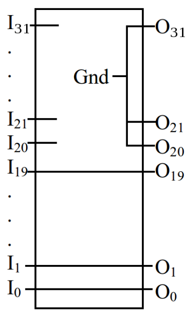
The next problem is designing the controller. We know that it will be a finite state machine that will first fetch a program instruction from the memory and then provide the correct operation sequences to execute that particular instruction. However at this stage we don't know what steps will be required, 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 BIT MASK circuit shown on the data path diagram and in detail on the diagram below. The BIT 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:

31 24	23 20	19 16	15 0
Opcode	**Rdest**	**Rscr**	**Unused**

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).

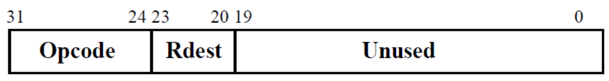
Two register instructions

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.

31 24	23 20	19 16	15 0
Opcode	**Rdest**	**Rscr**	**Unused**

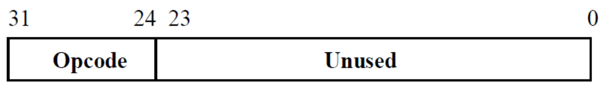
One register instructions

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

Instruction	Cycle	Transfers	Path
LOADINDIRCET Rdest, Rscr	E1	A←Rsrc	
	E2	MAR←A	
	E3	MDR←Memory	
	E4	Rdest←MDR	No masking
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 and carry 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 are the same. They either increment the program counter or do nothing. The NOP instruction does not require even one cycle to execute and has no associated register transfers.

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.