FIT1047 - Week 3

Central Processing Units

(part 2)

Recap

In the previous lecture we saw

- Basic CPU architecture
- MARIE assembly code
- Combinational circuits (adders, MUXes, decoders)
- ALUs

Overview

In this class, we add the remaining ingredients to be able to construct a complete CPU.

- Sequential Circuits
  - flip flops, registers, counters
  - memory
- Control
  - executing a program

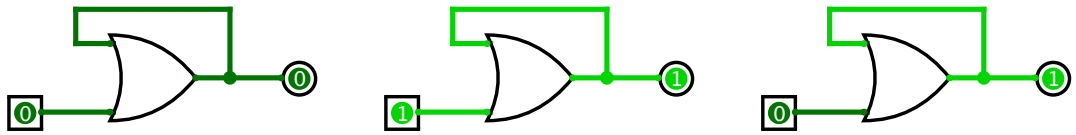
Sequential Circuits

Sequences

In a sequential circuit, the output depends on *past inputs*.

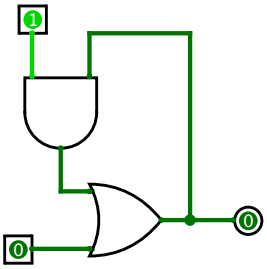
How can a circuit remember the past?

Feed the output back into the input!



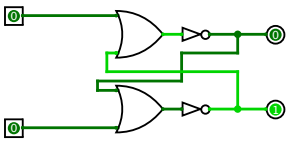
The output now represents the fact whether the input has *ever* been 1. When we toggle the input to 1 (circuit in the middle), the output of the OR gate becomes 1. Now that 1 is fed straight back into the OR, which means that the state of the input does not matter any longer, the output will stay 1 even if the input flips back to 0 (circuit on the right).

Let's add a switch to control the state.



When the switch is set to 1 (as in this picture), the circuit behaves exactly like the last one: as soon as we set the input to 1, the output will become 1 and stay like that. However, think about what happens if the output is one and we set the switch to 0: the AND gate now will output 0, so we can reset the stored state.

Can we implement a toggle?



This is called an **SR latch** (set-reset-latch).

The best way for you to understand how an SR latch works is to try it out in the simulator.

Let's take a look at the truth table for an SR latch. You will see that in addition to the inputs (S and R), we have a third column Q(t) which represents the state (the output) at time point t. The truth table then tells us what the state at the next time point Q(t+1) is going to be.

S	**R**	**Q(t)**	**Q(t+1)**
0	0	0	0
0	1	0	1

... continued on next page

S	R	Q(t)	Q(t+1)
1	0	0	0
1	1	forbidden	
0	0	1	1
0	1	1	1
1	0	1	0
1	1	forbidden	

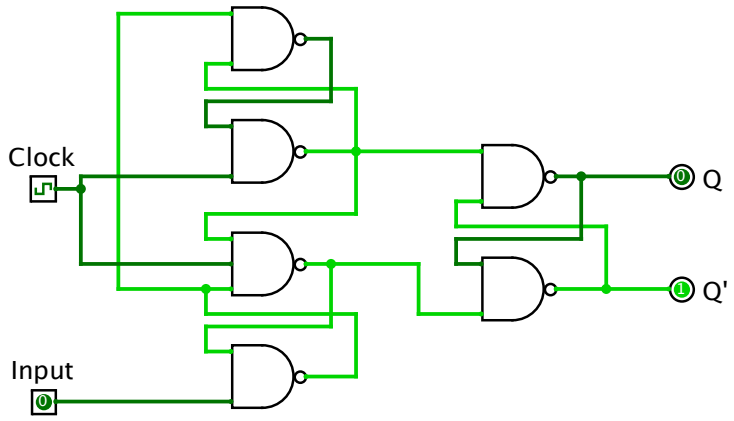
SR latches can store a single bit, based on the two inputs that allow us to set or reset the stored bit.

But digital circuits use a *single* input (bit).

We can add some circuitry so that instead of individual set and reset signals, we have a data input and a *clock signal*.

D flip-flop

- One input: the data to be stored
- One output: the data currently stored
- Plus a **clock**: state only changes on *"positive edge"*



Flip-flops actually produce two outputs: the "positive" output (Q) and its negation (Q'). We often only need Q and don't show Q'.

The *positive edge* of a clock cycle is the time during which the voltage rises from low to high. If you look at the circuit closely, you can see that it consists of three separate latches (in this case they are constructed using NAND gates instead of the NOR we used above). Without going into

too much detail, the left two latches make sure that the input to the latch on the right is never the forbidden state, even if the clock and the input bit do not change exactly at the same time.

To summarise, a D flip-flop is a *storage cell* for a single bit. We can now put several of these cells together to build registers.

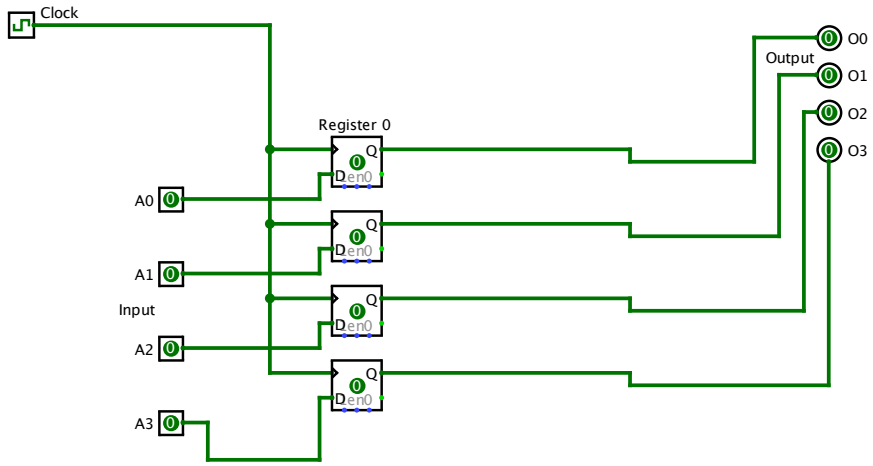
Registers

- Very fast memory inside the CPU
- Some special purpose registers
  - PC, IR, MBR, MAR (for MARIE)
- Some general purpose registers
  - AC (MARIE), AH/AL, BH/BL, CH/CL, DH/DL (x86)
- Fixed bit width
  - e.g. 16 bits in MARIE, 16/32 or 64 bits in modern processors

Register file

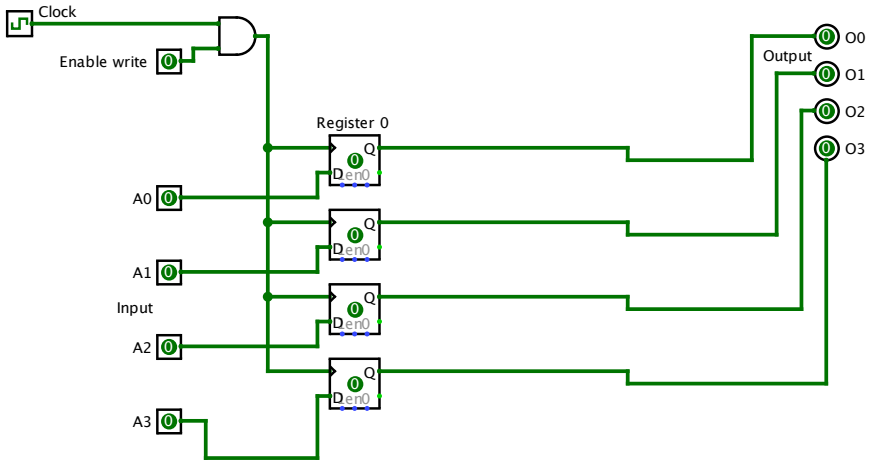
- Collection of registers
- Each implemented using n flip-flops (for n bits)
- n inputs and outputs
- Additional input: select register for writing
- Additional input: select register for reading

We will now construct a register file from the components we have seen so far, step by step.

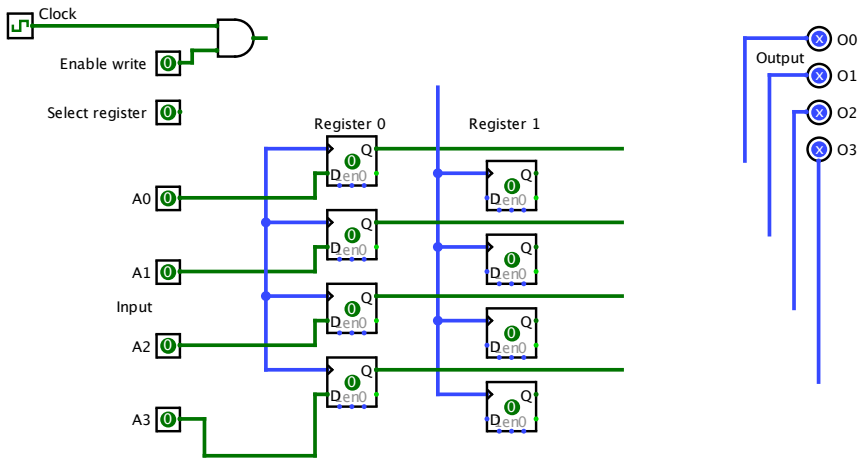


We use 4 D flip-flops to build a 4-bit register. Every clock cycle transfers the current input into the register, which stores it and outputs it.

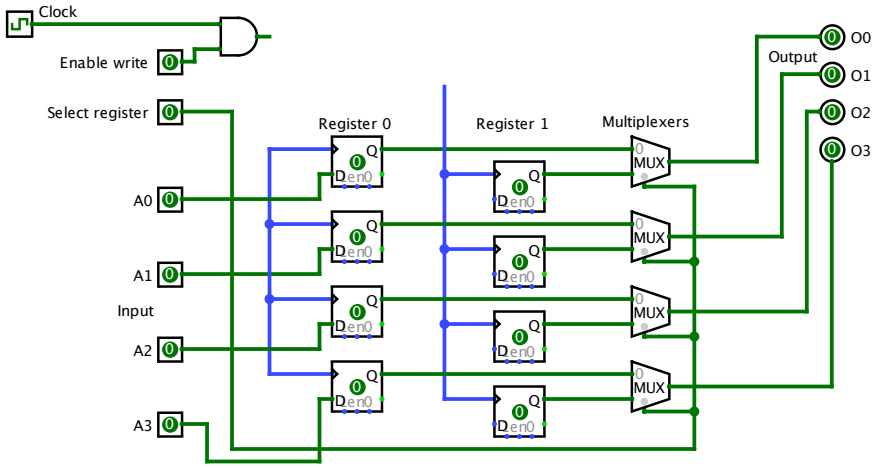
Remember that there is only one, central clock for the entire CPU (so that everything is synchronised). This would mean that the currently selected register is always overwritten during each clock cycle. But what if we don't want to write a new value at every clock cycle? To avoid that, the clock is *and*-ed with another input, the *write* signal. This is required so that the clock is only passed to the flip-flops when an operation actually needs to change the register contents.



Here we've added another input that lets us toggle whether we want to write. It is fed through an *and* gate with the clock. The output of the *and* is 0 if the write signal is 0, and is equal to the clock signal otherwise.



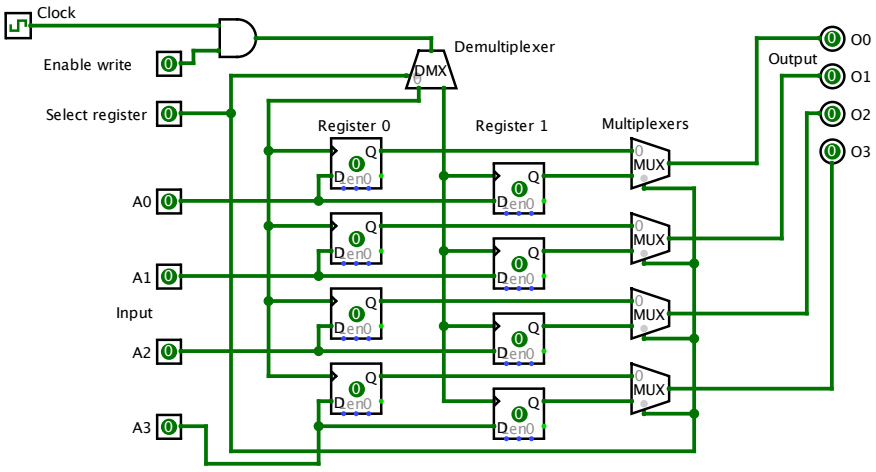
Now let's add another 4-bit register, to create a *register file*. We also add another input that lets us choose between the two registers.



Let's deal with the output first. Based on the selection input, we want to choose the output from register 0 or 1. That's exactly what a multiplexer does!

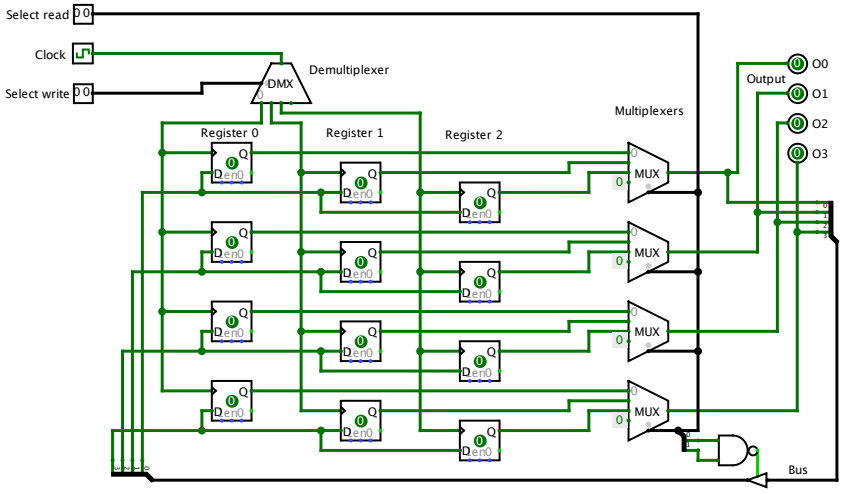
But how do we choose which register we write to?

Note how the clock is fed through a demultiplexer: only the flip-flops that are selected get the clock signal, so only those flip-flops will change their stored bits to the value of the input.



In reality, the register file of course doesn't have inputs and outputs that we can poke with a simulator. Both inputs and outputs are connected to the *data bus*, which connects the register file to other components such as the memory or the ALU.

The following is a (slightly) more realistic register file, with separate signals for selecting the read and write register.



This setup also allows us to *copy* the contents of one register into another one: if we set the read and write selectors to different registers, the data stored in the read register is put on the bus, and the write register stores the data from the bus. Remember that several operations in the MARIE instruction set require this behaviour (e.g. transferring the PC into MAR, or transferring the MBR into AC).

A real register file doesn't actually use individual D flip-flops but is implemented using more efficient technology (similar to very fast RAM).

Control

(the machine)

We have now seen components that can perform the basic computations (the ALU) and store results (the register file). The main component that's missing in order to construct a CPU is the logic that controls the ALU, the registers and the memory, based on the instructions of the program.

This component is the **control unit**. Its tasks are:

- Fetch - decode - execute cycle
- Implement the RTL code for each instruction
  - RTL = Register transfer language
- Generate **control signals** for individual circuits
  - opcode for the ALU
  - read/write register number for the register file
  - memory read/write

Timing

Let's take a look at the *Add X* instruction.

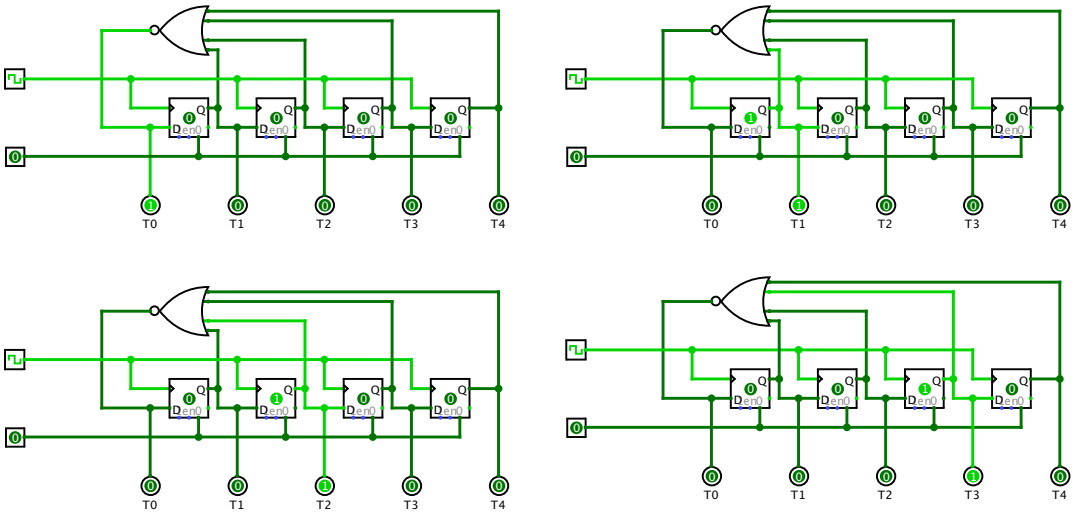
It needs to perform three distinct steps, according to its RTL definition.

- MAR $\leftarrow$ X
  - register read control: IR (X is in IR)
  - register write control: MAR
- MBR $\leftarrow$ M[MAR]
  - memory: read (always reads at address MAR)
  - register write control: MBR
- AC $\leftarrow$ AC + MBR
  - ALU opcode: addition (always reads AC and MBR)
  - register write control: AC

How do we generate that *sequence* of signals?

Add a **cycle counter**!

- n outputs T_0 to T_n
- Cycle through these outputs at each clock cycle.
- Has an input C_n to reset to T_0.
- Now we can write RTL using Boolean statements!



The cycle counter starts in state T_0 (top left). With each clock cycle, the previous timing signal is deactivated and the next one is activated.

Control signals

In order to define the signals that need to be activated for each RTL, statement let's define some notation.

Each register is assigned a unique identifier, which is simply a number between 0 and 7:

Register	Identifier	Identifier (binary)
MAR	1	001
PC	2	010
MBR	3	011
AC	4	100
IR	7	111

In order to switch the register file to read from and write to a certain register, it needs six control lines: P_2, P_1, P_0 are used for selecting the register to read from; P_5, P_4, P_3 select the register to write to.

So for example to write into the AC register, we need to put the bit pattern 111 on the P_5, P_4, P_3 control signals (switching them all on). To enable reading from the MBR, we would have to use the bit pattern 011 with the control signals P_2, P_1, P_0 (so P_2 will be 0, while P_1 and P_0 are 1).

The memory has two control signals, M_r for reading from memory and M_w for writing into memory. It always gets the address from the MAR register (there's a special "data path" from MAR to the memory), so we don't need to enable the MAR for reading when we want to use the memory.

In addition to controlling the register file and memory read/write signals, we also need to control the ALU. We use two control signals A_1 and A_0, where the bit patterns 01 mean addition, 10 means subtraction, and 11 means "clear" (setting the ALU output to 0).

We can now list the signals that need to be active for each RTL step. We will use the Add X instruction as an example.

Add X with signals

All instructions start with a *fetch*, where the instruction at the address pointed to by the PC is loaded into IR, and then the PC is incremented. This requires three steps (copy PC into MAR, load instruction from memory, increment PC) that are controlled by timing signals T_0, T_1 and T_2. The following steps for the Add instruction are:

Step 3: MAR $\leftarrow$ X

T_3 P_3 P_2 P_1 P_0

- read IR ($P_2 = 1$, $P_1 = 1$, $P_0 = 1$)
- write MAR ($P_5 = 0$, $P_4 = 0$, $P_3 = 1$)

Note: T_0, T_1, T_2 are used for the fetch cycle.

Step 4: MBR $\leftarrow$ M[MAR]

T_4 P_4 P_3 M_R

- Read memory (always reads at address MAR)
- write MBR ($P_5 = 0$, $P_4 = 1$, $P_3 = 1$)

Step 5: AC $\leftarrow$ AC + MBR

T_5 A_0 P_5 P_1 P_0

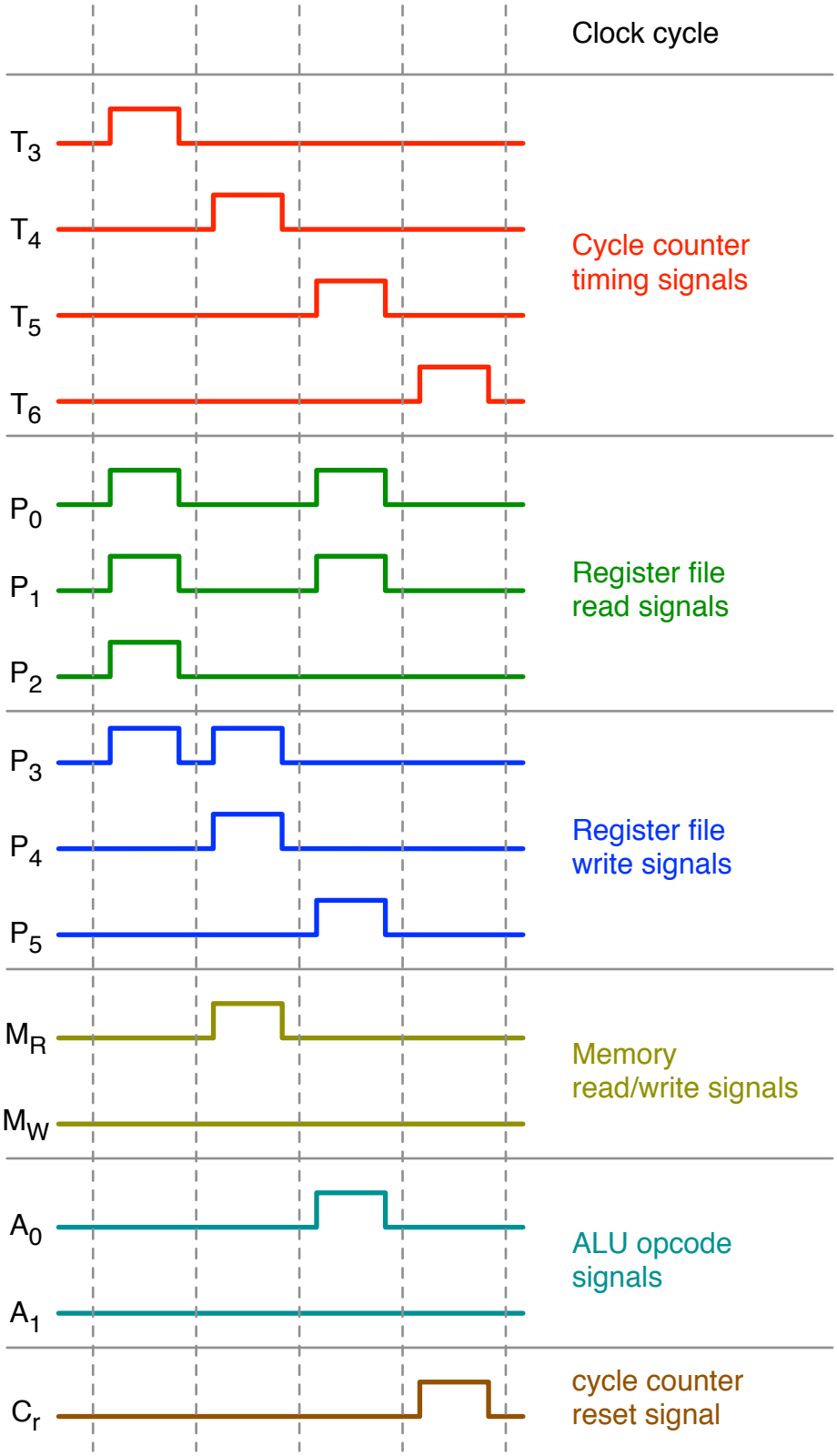
- ALU: addition ($A_1 = 0$, $A_0 = 1$)
- read MBR ($P_2 = 0$, $P_1 = 1$, $P_0 = 1$)
- write AC ($P_5 = 1$, $P_4 = 0$, $P_3 = 0$)

Step 6: AC $\leftarrow$ AC + MBR

$T_6\ C_r$

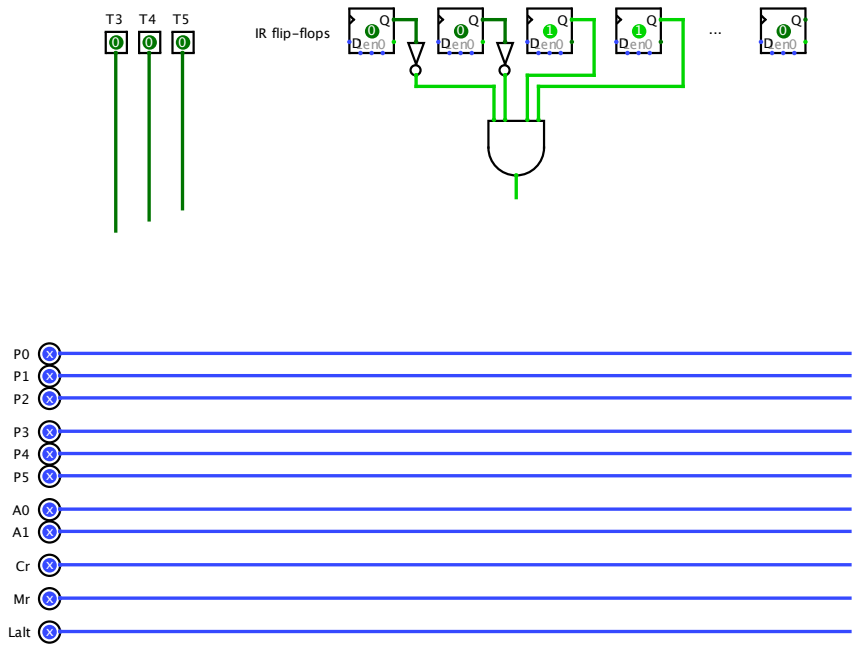
- reset cycle counter

The following figure shows a *timing diagram* for the execution of an Add instruction. Each column (labelled C_3 to C_6) represents one *clock cycle*. Each row corresponds to one of the control signals. Reading the diagram from left to right, you can see how the control signals change from one clock cycle to the next, according to the steps we saw above.


Clock cycle
T3
T4
T5
T6
Cycle counter timing signals
P0
P1
P2
Register file read signals
P3
P4
P5
Register file write signals
MR
MW
Memory read/write signals
A0
A1
ALU opcode signals
Cr
cycle counter reset signal

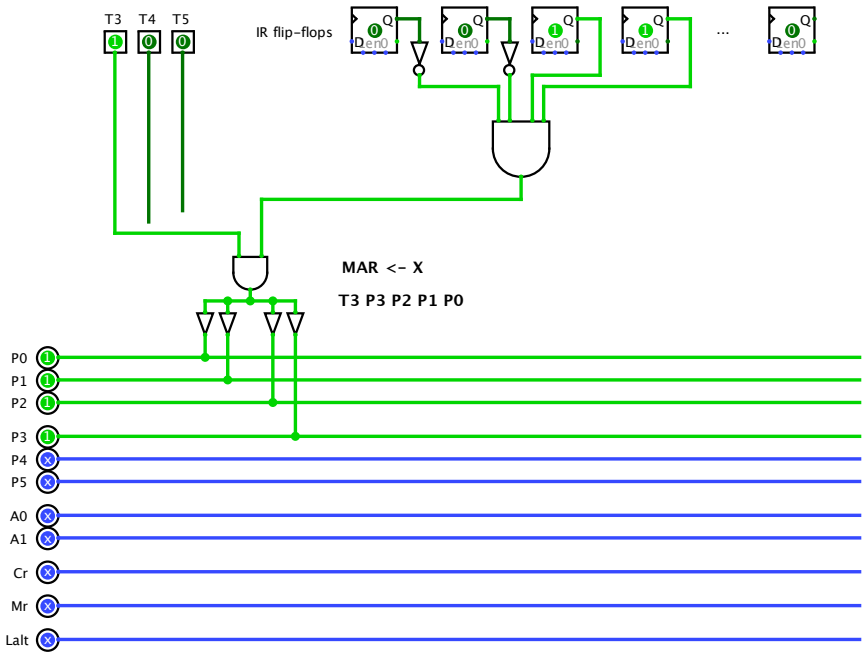
Now that we know which signals need to be generated for each instruction, we can implement a complete control circuit. In the following, we only show the circuit for the Add X instruction, but of course we would have to implement similar logic for every instruction in the instruction set.

Control circuit (Add X)



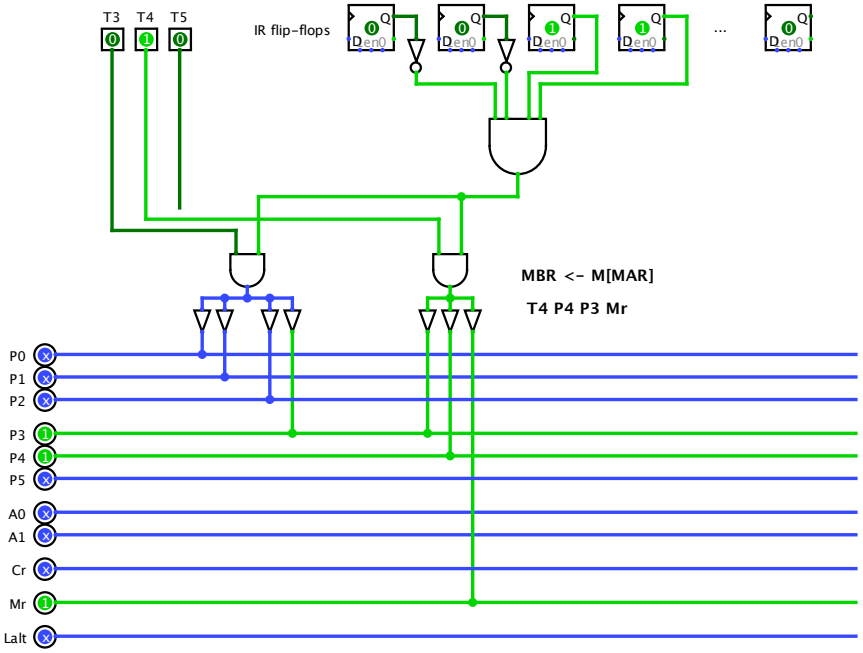
This circuit is decoding the opcode (the highest 4 bits in IR). This only shows the case for the Add instruction (0011). It simply uses two Not gates (for bits 15 and 14) and an And gate, whose output is only 1 if the instruction is an Add.

The lines at the bottom are the control signals, and the three inputs at the top come from the cycle counter. Now we need to combine the cycle counter with the instruction decoder to generate the correct signals for each cycle.

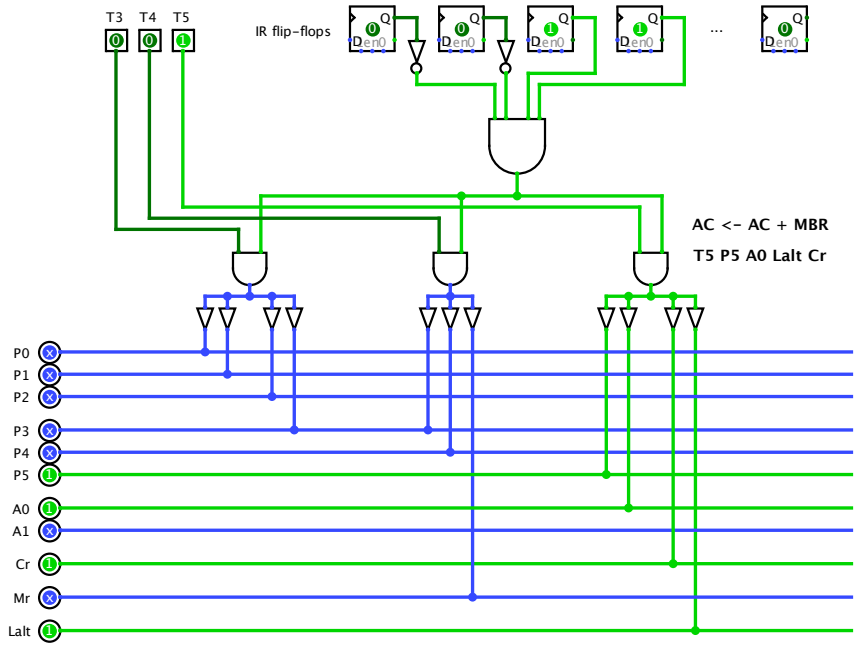


If the cycle counter T_3 is active (1) **and** the instruction is Add, then set control signals P_3, P_2, P_1, P_0.

In the next cycle, T_3 will become 0 again and T_4 will be 1.



If the cycle counter T_4 is active (1) **and** the instruction is Add, then set control signals P_4, P_3, M_R.



If the cycle counter T_5 is active (1) **and** the instruction is Add, then set control signals P_5, A_0, C_r, L_{alt}.

The little triangles (which look like Not gates without the circles at the outputs) are *buffers*, which make sure that signals only flow in one direction. Otherwise, e.g. in cycle T_3 the control signal P_3 is activated, but it would be directly connected via the And gate for T_4 to the control signals P_4 and M_r.

Hardwired Control

The control logic we have seen just now is called *hardwired*.

RTL for each instruction is implemented using gates.

Hardwired control has a few advantages and some really big disadvantages:

- Very fast!
- But really complicated to design.
- What if we want to add instructions?
- What if we made a mistake?

Designing a hardwired control circuit for a simple architecture like MARIE with only 16 instructions is already quite complicated (you can try it out!). Now imagine the control logic required for a CPU such as an Intel Core i7, which has hundreds of instructions!

Microprogrammed Control

Hardwired control logic is clearly unrealistic for large CPU projects. In order to make it easier to keep this complexity under control, we can use the fact that many instructions have similar steps in their RTL definitions. The solution is therefore:

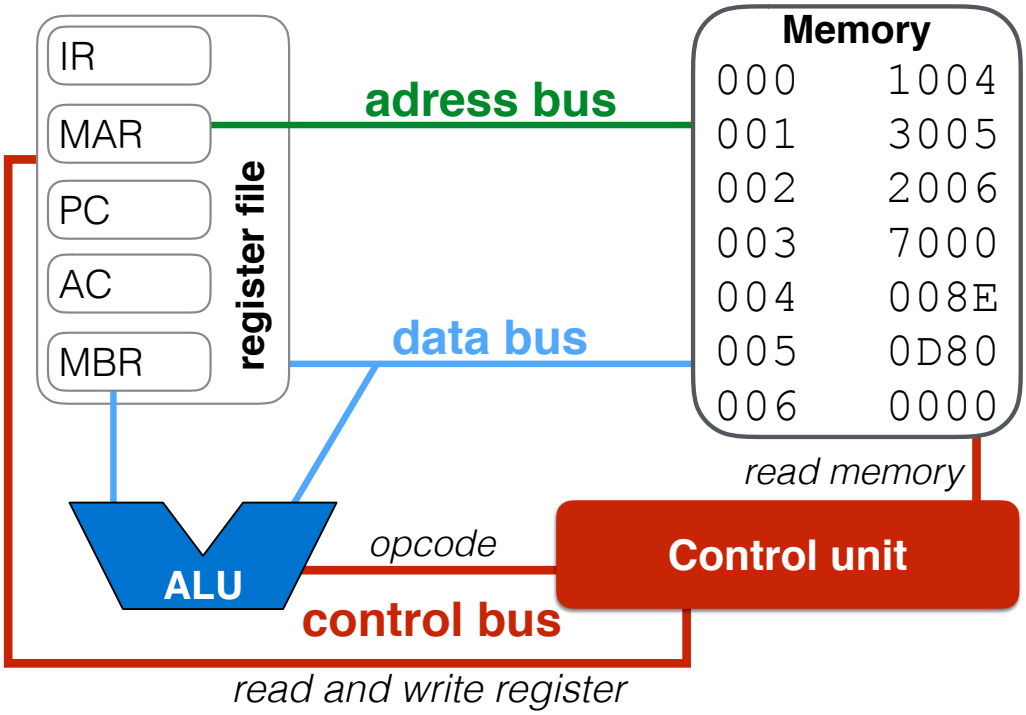
Break up instructions into **microoperations**.

- One microop per possible RTL instruction
  - Far fewer than instructions!
- Microops for each instruction stored in memory in the processor
- Execution slightly slower, but
  - Circuits for microops much simpler
  - Microcode can be updated

Modern PC processors (from the Intel x86 family) are micro-programmable, i.e., their microcode can be updated using software. This has allowed Intel to fix bugs in their processors even after they had been delivered to their customers!

Putting it all together

The following figure shows the complete *data paths* of the MARIE architecture.



Next lectures

- More MARIE instructions
- Memory
- Interrupts
- Input/Output