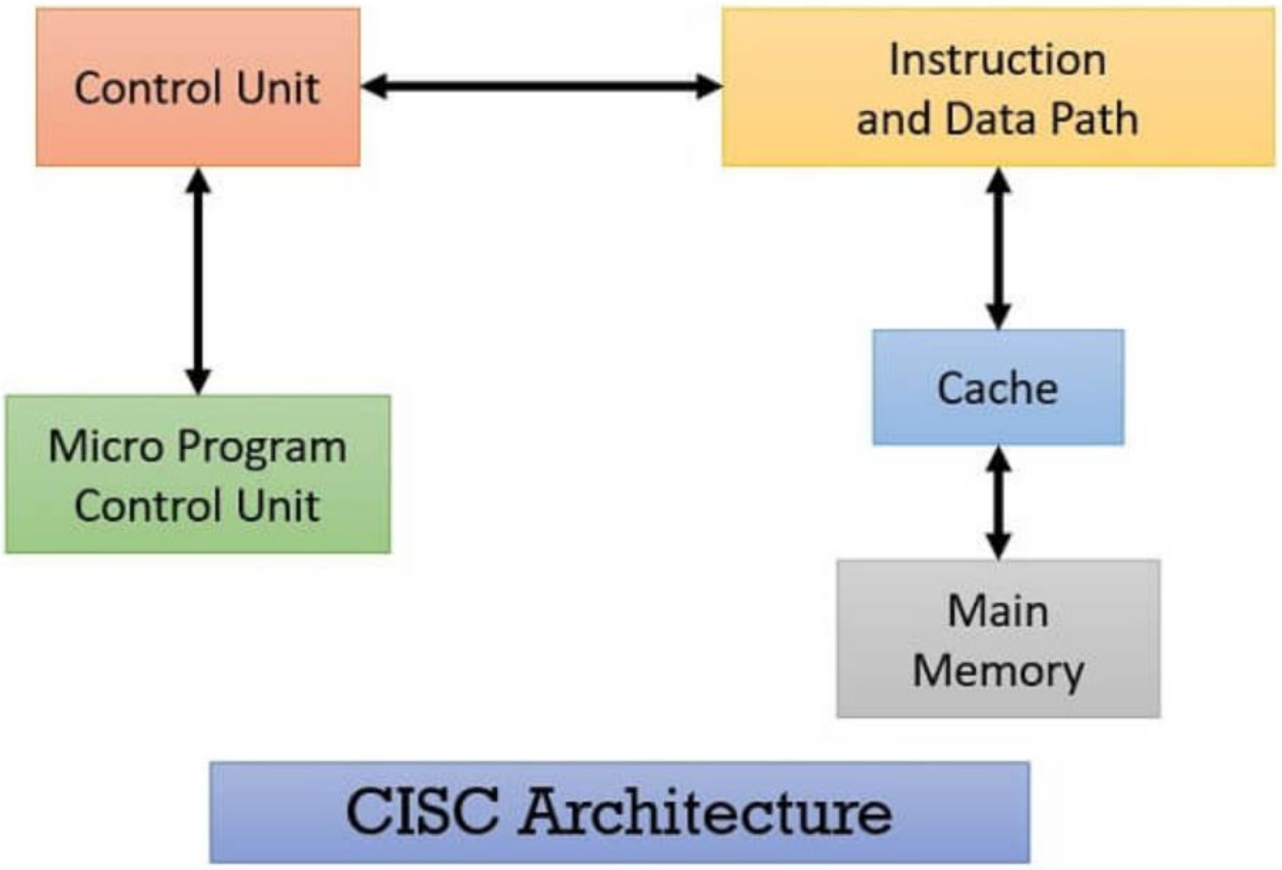
3. Microprogrammed computer



Microprogramming

- First computers were designed and built to execute simple machine language operations
  - Soon it was noticed, that many of these operations resembled each other
- Idea: separate a small very low-level group of basic commands and program each simple machine language command using these
  - This set of commands was named *microinstructions*
- Nowadays most modern computers execute microinstructions
- In order to do this, there needs to be an interpreter that decodes simple machine language to microcode
- Microcode is used to control the hardware of the computer
- Microprograms are often called *firmware*
  - Act as a link between the software and hardware

Structure of a microprogrammed computer

- In this course, we will familiarize ourselves with the concept of a microprogrammed computer using the following 16-bit example computer, which is comprised of the following components:
  - Registers
  - Memory
  - ALU
  - Control bus & three data buses
  - Five-stage clock
- The components of our example computer can be divided to four “units”
  - Each “unit” plays one part in operation
  - “Units” are not physical parts, as you will soon notice
- See the structure graph of the example computer in Moodle!

Unit 1: Control

- Microinstruction register (MIR)
  - 22-bit register, which includes the microinstruction that is currently in execution
  - Connected to control bus (CC), bits act as control bits for computer components
  - Control bit list can be found from slides 8-10
- Micro program memory (MPM), ROM
  - 22-bit register; storage unit of microinstructions
  - Read-only; instructions are “burned” on memory by the manufacturer
- Micro program counter (MPC)
  - 8-bit register, which tells the MPM address of the instruction in execution
  - Used in clock stages 4 (new search address) and 5 (new address given by DC3 saved to MPC)
- Clock (5 stages)
  - Gives a pulse (on their turn) for all five wires which activate bits in MIR

Unit 2: Memory and data transfer

- Memory data register (MDR)
  - 16-bit register, which is used to transport data from main memory to registers A…D
- Main memory (MM), RAM
  - Storage unit of machine coded programs and their data
  - 16-bit memory slots
  - Random-access memory (read and write); much slower than MPM
  - Data transfer from and to registers happens via MDR
- Memory address register (MAR)
  - 12-bit register; used as an address when using MM

- Analogy: MDR is a taxi that transports people (data) from their home (MM) to their workplace (registers A…D) and back; MAR is the taxi central that stores rides

Units 3 and 4: ALU & special registers

- Arithmetic-logic unit (ALU)
  - 16-bit full adder, which executes the calculations
  - Is capable of addition, subtraction using two’s complement (compl) and multiplication by two (shiftleft)
- Special registers A, B, C and D
  - 16-bit registers, which are meant for maintaining data (operands and interim results)
  - Can be read in clock stage 1 and written to in clock stage 2
  - Register A values can also be compared in clock stage 4
  - Comparisons available: $A < 0$ and $A = 0$

Buses & clock stages

- 22-bit control bus (CC)
  - MIR guides data transfer
- Three 16-bit data/address buses
  - DC1 and DC2 are used for providing inputs to ALU
  - DC3 is used to transfer the result from the ALU to the desired register
- In each clock stage, some of the following tasks can be done:
  - Stage 1: contents of MDR or number 1 is written in DC1 and content of the register A, B, C or D will be written in DC2 for arithmetic processing in ALU
  - Stage 2: The result from ALU is written from DC3 to some of the registers MAR, MDR, A, B, C or D
  - Stage 3: Contents of MDR are written to address given by MAR in MM – or contrariwise, contents of the MM address given by MAR will be written to MDR
  - Stage 4: New value of MPC will be calculated.
  - Stage 5: MPC gets a new value (automatically – no control bit!), and the microinstruction specified in MPC will be transferred to MIR register

Control bits

- Different actions can only be performed during certain clock stages
- Control bits in clock stage 1:
  - c1 = write contents of register A to bus 2 (DC2)
  - c2 = write contents of register B to bus 2 (DC2)
  - c3 = write contents of register C to bus 2 (DC2)
  - c4 = write contents of register D to bus 2 (DC2)
  - c5 = write 1 to bus 1 (DC1)
  - c6 = write contents of MDR to bus 1 (DC1)
  - c7 = complement contents of bus 1 (DC1) (for subtraction purposes)
  - c8 = result of addition in bus 3 (DC3) is multiplied by 2 (shiftleft; see slide 13)

Control bits

- Control bits in clock stage 2:
  - c9 = read contents of bus 3 (DC3) to register A
  - c10 = read contents of bus 3 (DC3) to register B
  - c11 = read contents of bus 3 (DC3) to register C
  - c12 = read contents of bus 3 (DC3) to register D
  - c13 = read contents of bus 3 (DC3) to MDR
  - c14 = read contents of bus 3 (DC3) to MAR

- Control bits in clock stage 3:
  - c15 = read contents of MM address given by MAR to MDR
  - c16 = write contents of MDR to MM address given by MAR

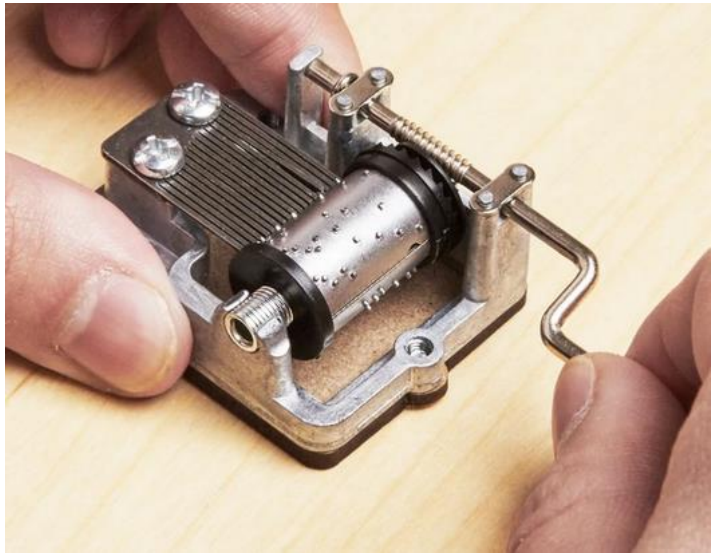
Control bits

- Control bits in clock stage 4:
  - c17 = write 1 to bus 1 (DC1)
  - c18 = write 8 most significant bits of MIR to bus 1 (DC1)
  - c19 = if the contents of A is zero, write 1 to bus 1 (DC1); else, write 2 to bus 1 (DC1)
  - c20 = if the most significant bit of A is 1, write 1 to bus 1 (DC1) ; else, write 2 to bus 1 (DC1)
  - c21 = write 4 most significant bits of MDR to bus 1 (DC1)
  - c22 = write contents of MPC to bus 2 (DC2)

- Clock stage 5 contains no control bits

Microprogramming

- Microprogrammed computer executes a program stored in main memory
- Each task is performed by executing a microprogram stored in MPM
- Fixed list of operations – the desired ones are activated via microinstructions
  - Analogy: music boxes – pins on the drum activate certain notes
  - When the drum rotates, a certain melody is heard
- Microprogramming enables changes in order of operations
  - Conditional skipping of commands (skip)
  - Unconditional jumps (jump)
  - …these aren’t possible in music boxes



Microprogramming

- Basically microprogramming is regular programming; operations are very simple
- Each microinstruction contains 5 sub-instructions
  - 1 sub-instruction per each clock stage
  - Stage 1: right side of placement sentence (what is being calculated?)
  - Stage 2: left side of placement sentence (where the result is put?)
  - Stage 3: memory handling (MDR to MM or contrariwise)
  - Stage 4: calculate where to proceed next
  - Stage 5: move to next microinstruction
- Each microinstruction performs either a placement sentence or branching

Example 1: Calculation of 1-bits

- Microprogram, which calculates how many 1-bits the word found in MM in address given by register D has, and saves the result to register C
- Our computer is only capable of performing comparisons in register A, so we need to work within that register
- We only have comparisons A < 0 and A = 0 available
  - Due to two's complement method, if the first bit of our word is 1, it's negative
  - So, if A < 0 is true, the first bit is 1
  - There is no way to investigate the following bits one by one, so we have to modify the word by moving it one bit at a time to the left
- This can be done via the shiftleft (x2) operation:
  - 1st bit is discarded, others move 1 step left
  - Last bit it set to 0

1	0	0	1	1	0	0	1

→

0	0	1	1	0	0	1	0

Example 1: Pseudo-code algorithm

```
C:=0
A:=get(D)
WHILE A <> 0 DO
     IF A < 0 THEN
          C:=C+1
     ENDIF
     shiftleft(A)
ENDWHILE
```

- This isn’t possible to execute in a microprogrammed computer, because there are
  - Subprograms (get)
  - Loops (while)

Example 1: Microprogram?

- Replace subprograms with microinstructions and loops with jumps:

```
      C:=0
      MAR:=D
      MDR:=(MAR)
      A:=MDR
if:   IF A = 0 THEN jump pass ENDIF
      IF A < 0 THEN C:=C+1 ENDIF
      shiftleft(A)
      jump if
pass:
```

- This is close, but the jump is inside an if clause. Conditional jumps are not ok.

Example 1: Microprogram

- Change to conditional skips and unconditional jumps

```
        C:=0
        MAR:=D
        MDR:=(MAR)
        A:=MDR
 if:    skip A = 0
        jump pass
        skip A < 0
        C:=C+1
        shiftleft(A)
        jump if
pass:
```

If skip condition is true

If skip condition is false

Example 1: Symbolic microprogram

1: 0+0→C; ;1+MPC →MPC

2: 0+D →MAR; (MAR) →MDR; 1+MPC →MPC

3: MDR+0 →A; ; 1+MPC →MPC

4: ; ; (A=0)+MPC →MPC

5: ; ; 1010_2 →MPC

6: ; ; (A<0)+MPC →MPC

7: 1+C →C; ; 1+MPC →MPC

8: (0+A)x2 →A; ; 1+MPC → MPC

9: ; ; 100_2 →MPC

10:

Example 1: Symbolic microprogram

```
1:   0+0→C; ;1+MPC →MPC
2:   0+D →MAR; (MAR) →MDR; 1+MPC →MPC
3:   MDR+0 →A; ; 1+MPC →MPC
4:   ; ; (A=0)+MPC →MPC
5:   ; ; 1010₂ →MPC
6:   ; ; (A<0)+MPC →MPC
7:   1+C →C; ; 1+MPC →MPC
8:   (0+A)x2 →A; ; 1+MPC → MPC
9:   ; ; 100₂ →MPC
10:
```

Clock stage 1&2
Clock stage 3
Clock stages 4&5

Example 1: Binary microprogram

- Only 1-bits marked (others are zero)

Address	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1											1						1					1
2				1										1	1		1					1
3						1			1								1					1
4																			1			1
5					1		1											1				
6																				1		1
7			1		1						1						1					1
8	1							1	1								1					1
9						1												1				

← Control bits c1…c22

Clock stage 1 = c1…c8
Clock stage 2 = c9…c14
Clock stage 3 = c15…c16
Clock stage 4 = c17…c22
(Clock stage 5 contains no control bits.)

Example 2: Multiplication

- ALU of our microprogrammed computer does not have multiplication operation
- So, we need to implement this in another way
- The simplest way to perform multiplication is to convert it to a summation:
  - Sum together multiplier pcs of multiplicands

$$5 \cdot 3 = \underbrace{3 + 3 + 3 + 3 + 3}_{\text{5 pcs}}$$

- How could we write a microprogram algorithm that does this?

Example 2: Multiplication

- Algorithm:

```
MDR:=0
WHILE A <> 0 DO
      MDR:=MDR+C
      A:=A-1
ENDWHILE
```

- While-loop is not supported in microprograms, so it needs to be replaced with skip/jump commands

Example 2: Multiplication

- Microprogram algorithm:

```
        MDR:=0
1:      skip A=0
        jump 5
        MDR:=MDR+C
        A:=A-1; jump 1
5:
```

If skip condition is true → jump 5

If skip condition is false → MDR:=MDR+C

- Binary program:

Address	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
0													1				1					1
1																			1			1
2						1		1										1				
3			1			1							1				1					1
4	1				1		1		1								1					

Example 2: Algorithm efficiency

- The repetition phase of this algorithm comprises of 3 microinstructions (1,3,4)
- Our computer is 16-bit, so the maximum value of multiplier is $2^{16} - 1$
- Therefore, the maximum number of microinstructions to be performed is approx.

$$(2^{16} - 1) \cdot 3 = 196\,605$$

- Each microinstruction takes 5 clock cycles, so the maximum number of clock cycles needed is about 1 million
- CPU's clock speed defines the elapsed time (for example, 10 MHz → 0.1 seconds)
- Can the multiplication be performed quicker by using a more clever algorithm?

Example 2: Grade school multiplication algorithm

- Let’s take a moment to remember how we performed multiplications without a calculator in elementary school:
  - 7*3 = 21; write 1 and carry the 2
  - 7*5 = 35, add the carried 2 = 37; write 7 & carry 3
  - 7*4 = 28, add the carried 3 = 31
  - Move one step left, start from tens: 2*3 = 6
  - 2*5 = 10; write 0 and carry the 1
  - 2*4 = 8; add the carried 1 = 9
  - Move one step left, start from hundreds: 1*3 = 3
  - 1*5 = 5
  - 1*4 = 4
  - Sum all results together

```
   453
  *127
  ----
  3171
  906
+453
------
 57531
```

Example 2: Grade school multiplication algorithm

- The same algorithm works for binary numbers, too!
  - Multiplicand is added to the sum 0 or 1 times
- Due to the limitations set by our computer, we have to perform some modifications:
  - Because only the most significant bit (1st bit on the left) can be examined separately, the partial sums (multiplicand*multiplier) have to be formed in reverse order – from left to right
  - Partial sums are added to the total sum right after they've been formed (not mandatory, but saving partial sums to main memory kills performance advantages)
  - Because we're moving from left to right, the new partial sum should be moved right before adding it to the sum; we can't do this, so we move the old sum to the left instead (shiftleft)

```
  Multiplicand      000101
* Multiplier        000110
--------------------------
                    000000
                   000101
                  000101
                 000000
                000000
+              000000
--------------------------
               00000011110
```

Example 2: Grade school multiplication algorithm

- As a result of these modifications, the multiplication looks like this:

- Because the word size is 16 bits, this multiplication has to be done 16 times in our computer
- So, there will be 16 partial sums
- How to tell our computer when to stop the multiplication?
- The most natural way would be to use a counter which is formatted to have a value 16 in the beginning and then reduce it by 1 each round
- In microcode, we can’t set values this easily, so we’ll do this another way round: a fake 1 is added to the end of our multiplier
- For this last 1, the partial sum is not calculated

```
  0101
 *0110
 -----
  0000        1st partial sum
+  0101       2nd partial sum
 ------
  00101
+   0101      3rd  partial sum
 -------
  001111
+    0000     4th partial sum
 --------
  0011110     Total sum
```

Example 2: Grade school multiplication algorithm

```
        MDR:=0
        IF A < 0 THEN                               (*1st bit of A is 1)
                MDR:=MDR+C
        ENDIF
        shiftleft(A)
        A:=A+1                                      (*fake 1*)
loop:   IF A < 0 THEN                               (*multiplier bit is 1*)
                shiftleft(A)                        (*move multiplier left*)
                IF A = 0 THEN                       (*previous multiplier 1st bit was the last 1*)
                        jump stop                   (*because the last 1 is the fake 1*)
                ELSE
                        shiftleft(MDR)              (*move the old sum left*)
                        MDR:=MDR+C                  (*perform addition of sum and partial sum C*)
                        jump loop
                ENDIF
        ELSE                                        (*multiplier bit is 0*)
                shiftleft(MDR)                      (*move old sum left*)
                shiftleft(A)                        (*move multiplier left*)
                jump loop
        ENDIF
stop:
```

Example 2: Grade school multiplication, symbolic microprogram

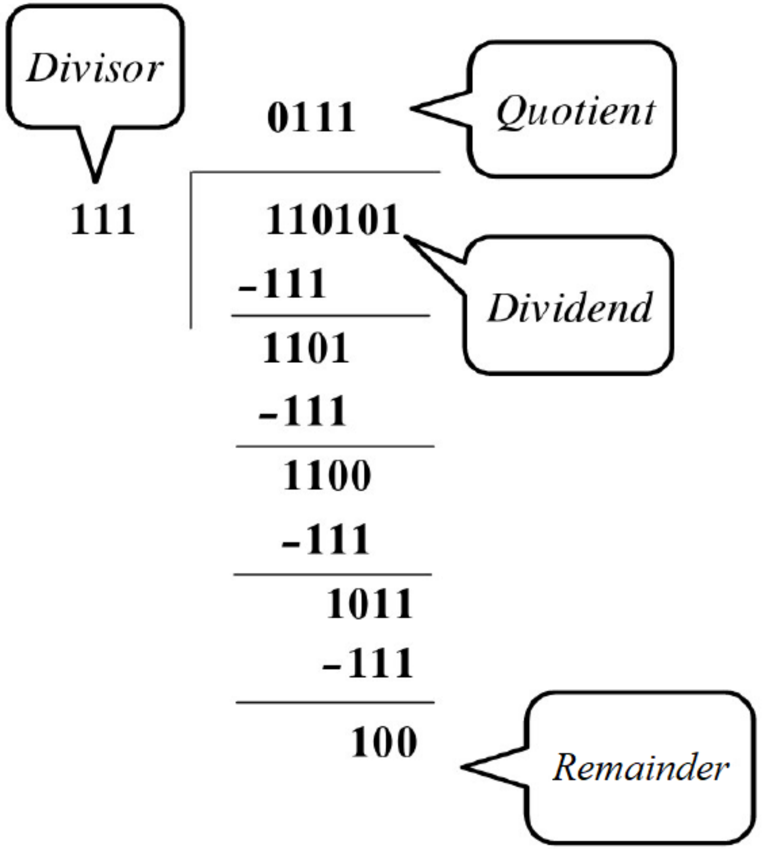
Address	Microinstruction	Explanation
0	0+0 →MDR; ; (A<0)+MPC →MPC	Set MDR=0. Is 1st bit of A 1?
1	MDR+C →MDR; ; 1+MPC →MPC	Yes; add C to MDR
2	(0+A)x2 →A; ; 1+MPC →MPC	Move A to left
3	1+A →A; ;1+MPC →MPC	Add fake 1 as the last bit of A
4	; ; (A<0)+MPC →MPC	Is the 1st bit of A 1?
5	; ; 1001+0 →MPC	Yes; jump to address 9
6	(MDR+0)x2 →MDR; ; 1+MPC →MPC	No; move MDR to left
7	(0+A)x2 →A; ; 1+MPC →MPC	Move A to left
8	; ; 100+0 →MPC	Jump to beginning of loop (4)
9	(0+A)x2 →A; ; (A=0)+MPC →MPC	Move A to left; is A=0?
10	; ; 1110+0 →MPC	Yes; stop (jump to address 14)
11	(MDR+0)x2 →MDR; ; 1+MPC →MPC	No; move MDR to left
12	MDR+C →MDR; ; 1+MPC →MPC	Add C to MDR
13	; ; 100+0 →MPC	Jump to beginning of loop (4)

Example 2: Grade school multiplication, algorithm efficiency

- The microprogram of 1st option was much simpler; why is this better?
- The repetition phase of this algorithm comprises of 4 (4,6,7,8) or 6 microinstructions (4,5,9,11,12,13)
- The initiation is 4 microinstructions (0,1,2,3) and the last repetition is 4 microinstructions (4,5,9,10)
- Because the computer only has to perform 16 multiplications, this multiplication algorithm needs in total 4+15*6+4 = 98 microinstructions
  - Quite a lot less than the original maximum of 196 605!
- Quick calculation tells us that this algorithm can be 2000 times faster
- Also, the time needed to complete the multiplication is not much dependent on the magnitude of multiplier (unlike the previous algorithm)

Division of binary numbers

- Division of binary numbers can be implemented in a similar fashion as division of decimal numbers
- The method is called “shift-and-subtract”
- More efficient ways have also been invented
    - Fast division algorithm (and its variations)



Binary division:
1. Align the divisor Y with the most significant end of the dividend. Let the portion of the dividend from its MSB to its bit aligned with the LSB of the divisor be denoted X.
2. Compare X and Y.
    a) If X >= Y, the quotient bit is 1 and perform the subtraction X-Y.
    b) If X < Y, the quotient bit is 0 and do not perform any subtractions.
3. Shift Y one bit to the right and go to step 2.

Summary

- Information that has been stored in the memory of a micro-programmed computer affects the operation of the computer, so it is capable of executing different algorithms
- The structure (logic circuits made of logic gates) of the micro-programmed computer defines the operations that the computer includes; all other functions must be performed via microprograms
- Presentation methods of microprograms:
    - Pseudocode
    - Symbolic microprogram
    - Binary microprogram

Thank you for listening!