# CEG 2136 Cheat Sheet

# 1 Chapter 1

This chapter is mostly a review of ITI 1100.

# 2 Chapter 2

## 2.1 Multifunction Circuits

This is a circuit that preforms more than one function. For example, a circuit can store if $S = 0$, and count up if $S = 1$.

| A1A0 | Function |
|---|---|
| 00 | Store |
| 01 | Count up |
| 10 | Load |
| 11 | Clear |

Next we will make a state table with the following columns (assuming we are doing a D flip flop which is easy and 4 bits)

| A1A0 | Q3Q2Q1Q0 | Q3'Q2'Q1'Q0' | D3D2D1D0 |
|---|---|---|---|
| *operation* | *current state* | *next state* | *flip flop inputs* |

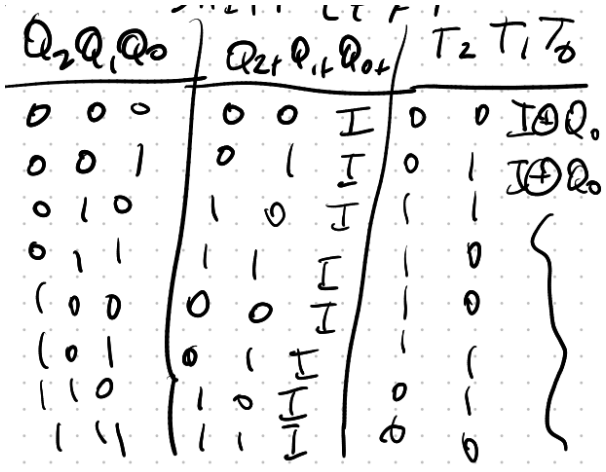
Then we can create all 4 flip flops, and then put a MUX in front of each of them with the correct operations (with A1A0 as selectors)

**Ex. Design a multifunction register as specified in the table using T flip flops.**

| **c1** | **c0** | **Function** |
|---|---|---|
| 0 | 0 | Store registers contents |
| 0 | 1 | Left Shift (serial input connected to input I) |
| 1 | 0 | Count Down |

**Store:** The store is easy, since T ff, we just feed in 0.

**Left Shift** We are not sure about this one. It would be easy with a D ff, but we are using a T. Whenever we are unsure, we create a table to visualize it.



For the T2 and T1, it is self explanatory. For T0, we need the XOR because again we are using T ffs not D ffs. It makes sense, just trust me.

Now for the T2 and T1, we need an equation so use 2 k maps to get:

$$T_1 = Q_0 \oplus Q_2$$
$$T_2 = Q_2 \oplus Q_1$$
$$T_0 = I \oplus Q_0$$

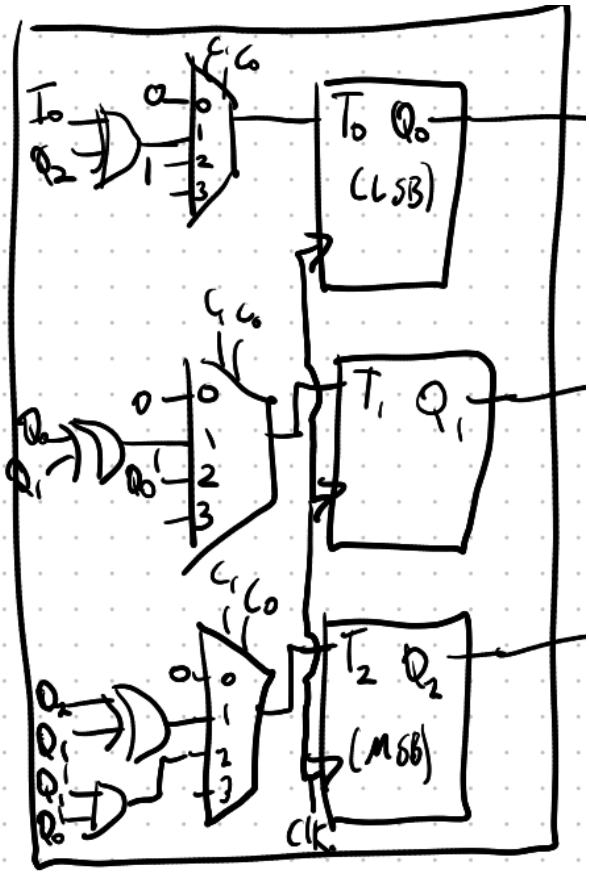
YAY!! Now we just need 1 more ops.

**Count Down:** This is the same process as the shift left. We just need to create the state table, and then find the values for T2T1T0 using K maps:

$$T_0 = 1$$
$$T_1 = Q_0'$$
$$T_2 = Q_1'Q_0'$$

Now we are done. We know how to do each of the 3 operations, we just need to *choose* between them. Since we see **choose**, we know MUX.

We can ignore the last input of our 4x1 muxs since we do not need it.




Most cases are not this bad since we use D flip flops.

For example, for the left shift using a D, we just put D2 as D1, and D1 as D0, and then D0 as I. SIMPLE!!

### 2.2 Memory

If we have $k$ address lines in a memory chip and $n$ data output lines, it means we can store $2^k$ $n$ bit words in that memory chip.

## 3 Chapter 3

### 3.1 2s Compliment in Binary

To get the 2s compliment of a number in binary, we flip EACH bit (so 1 becomes 0, 0 becomes 1) and then we add 1 to that number.

Note that when we are using signed numbers where 0 is + and 1 is -, then the 2s compliment of a positive number is itself.

### 3.2 Overflow

With signed numbers, overflow occurs when the carry into the sign bit and carry out of the sign bit are different. So:

$$OVERFLOW = C_{sign-in} \oplus C_{sign-out}$$

**Ex.**

| | | |
|---|---|---|
| Carry | 1 0 0 0 | |
| -3 | 1101 | |
| + -6 | +1010 | |
| -9 | 10111 | = +7? |

Here overflow occurs since the bit into carry 0 is not the same as the bit out of carry 1.

### 3.3 IEEE Floating Point

This is a standerdized way of representing floating bit numbers.

We have 3 parts to the 32 bit number to represent a binary number $n$.

1. Sign (1 bit)
2. Exponent (8 bits) - 127 + The exponent the fractional part needs to be raised to to get $n$
3. Mantissa (23 bits) - The fractional part of the normalized number $n$

**Ex. Express $(36.5625)_{10}$ as a 32 bit FP number using the IEEE standard.**

The value in binary is 100100.1001 and normalized is $1.001001001 \times 2^5$.

1. Sign: 0
2. Exponent: 5+127 = 132 = 10000100
3. Mantissa: 00100100100000000000000

So the 32 bit number is: 0 10000100 00100100100000000000000 (without the spaces ofc)

# 4 Chapter 4

An operation executed in *one* clock cycle is called a **micro operation**.

## 4.1 Arithmetic Operations

These are things such as adding, subtracting, and really anything that uses an adder block.

## 4.2 Logic Operations

These are things that are not arithmetic but are still done such as AND, OR, SHIFTS, etc.

A circular shift will shift everything left or right and the spare bit will be taken from the other side.

A logical shift will shift everything left or right and the spare bit will be always a 0.

An arithmetic shift will shift everything left or right and the spare bit if right will be the same as the original MSB, and if left will be 0.

Right Arithmetic shift is division, and Left Arithmetic shift is multiplication.

# 5 Chapter 5

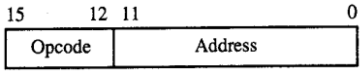
For this entire chapter (and course) we consider a computer to have 16 bits. So each instruction has 16 bits, and each register can hold at most 16 bits (some are less though).

## 5.1 Instruction Codes

Each instruction is a group of 16 bits that instruct the computer what to do.

An instruction is divided into the **opcode** (*First 4 bits*) and the **address** (*Last 12 bits*)

The CU receives the instruction from memory, interprets the **opcode**, then issues a sequence of control signals to some of the registers, and maybe the ALU.




### 5.1.1 MRI

Memory Reference Instructions (MRI) specify a **memory location using bits 0-11**.

For example, ADD ___ adds the content of the accumulator to the specified memory location.

### 5.1.2 RRI

Register Reference Instructions (RRI) start with a 7 using bits 12-15, and then for bits 0-11, it uses those to also choose the instruction rather than a memory location.

For example CMA will complement the accumulator, it uses **no external memory address**, just the accumulator.

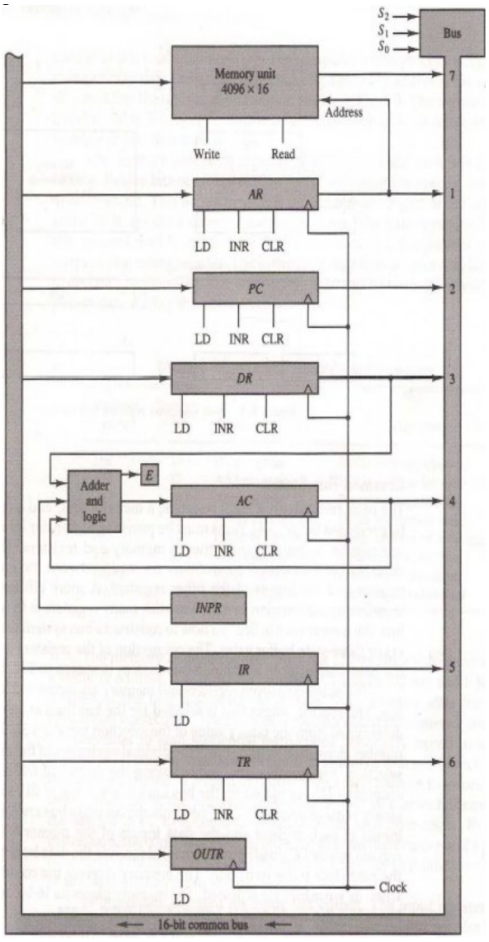
### 5.1.3 IOI

Input Output Instructions (IOI) work similar to RRI, but these are for input and outputs.

## 5.2 Registers and BUS

The computer has the following registers:

- AR: Address Register stores the memory address from bits 0-11 of the instruction
- PC: Program Counter stores the memory address of the next instructions
- DR: Data Register stores some data we want to use with the ALU
- AC: Accumulator which is the main register since it is the output of the ALU
- IR: Stores the current instruction code
- TR: Temporary Register is sometimes used for temporary storage
- OUTR/INPR: Holds Input or Output character.




Most of these registers can connect to the bus assuming the bus has that register selected.

**Ex.** What is the state of the computer to send data from memory location 100 (in AR) to the DR?

First we need to get the information into the bus. To do this, we need to set S2, S1, S0 to be 7, so 111.

The memory will read from the AR *only if READ=1*, so we set READ to 1.

Now we load the DR (set LD of DR to 1).

If we wanted, next cycle we could get it into the AC

## 5.3 Direct vs Indirect

If we say M[100], this means get the data located **in memory at location 100**. This is direct.

If we say M[M[100]], this means go to memory at location 100, and there will be a memory address. We want the **data at the memory address located at M[100]**. This is indirect.

## 5.4 Basic Computer Instructions

We have the following instruction list for the basic computer.

The first part are MRI, the second part RRI, the third part IOI.

Note that I refers to indirect (1) or direct (0).

| Symbol Symbole | Code Hex code | | Description |
|---|---|---|---|
| | $I = 0$ | $I = 1$ | |
| AND | 0xxx or 8xxx | | AND $M$ to $AC$ |
| ADD | 1xxx or 9xxx | | Add $M$ to $AC$, carry to $E$ |
| LDA | 2xxx or Axxx | | Load AC from $M$ |
| STA | 3xxx or Bxxx | | Store AC in $M$ |
| BUN | 4xxx or Cxxx | | Branch unconditionally to $m$ |
| BSA | 5xxx or Dxxx | | Save return address in $m$ and branch to $m + 1$ |
| ISZ | 6xxx or Exxx | | Increment $M$ and skip if zero |
| CLA | 7800 | | Clear AC |
| CLE | 7400 | | Clear $E$ |
| CMA | 7200 | | Complement AC |
| CME | 7100 | | Complement E |
| CIR | 7080 | | Circulate right $E$ & $AC$ |
| CIL | 7040 | | Circulate left $E$ and AC |
| INC | 7020 | | Increment AC, |
| SPA | 7010 | | Skip if $AC$ is positive |
| SNA | 7008 | | Skip if $AC$ is negative |
| SZA | 7004 | | Skip if $AC$ is zero |
| SZE | 7002 | | Skip if $E$ is zero |
| HLT | 7001 | | Halt computer |
| INP | F800 | | Input data and clear flag |
| OUT | F400 | | Output data and clear flag |
| SKI | F200 | | Skip if input flag is on |
| SKO | F100 | | Skip if output flag is on |
| ION | F080 | | Turn interrupt on |
| IOF | F040 | | Turn interrupt off |

### 5.4.1 ISZ

ISZ ___ will increment the memory location by 1, and then if it is 0, it will skip the next instruction.

### 5.4.2 BSA/BUN

BUN ___ will set the PC to the given memory location.

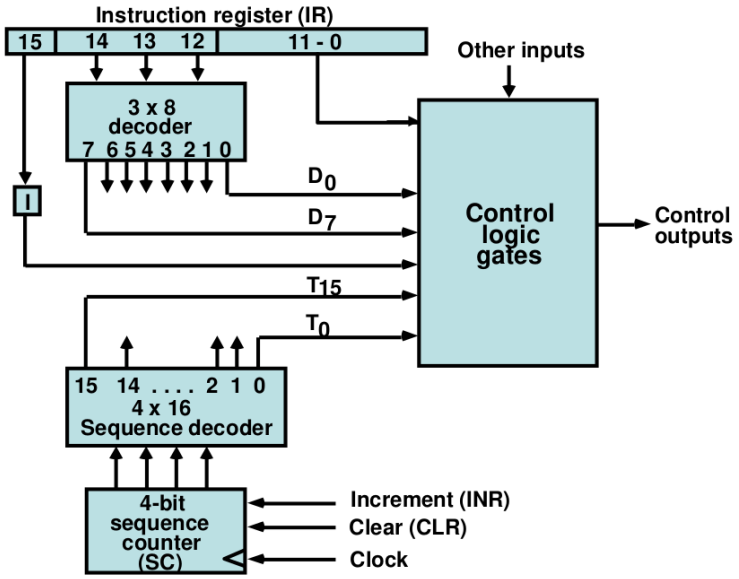
BSA ___ will set the PC to the given memory location and save the current PC to memory location 0.

## 5.5 Timing

For timing stuff, we are working inside of the CU.

Note that the CU works off of lots of binary bits such as **T0-T15 (timing states), D0-7 (decoded opcode), IR0-IR11 (address of instruction)** and it outputs all the control signals for the bus, and registers (such as **DR Load, Memory READ**).

We have a state counter (SC) which has 16 states and starts at 0, and each clock cycle increments.




The instruction cycle has the following 4 phases:

1. Fetch instruction from memory T0, T1
2. Decode the instruction T2
3. Read the address from memory if indirect address
4. Execute the instruction

We have the following instruction cycle.

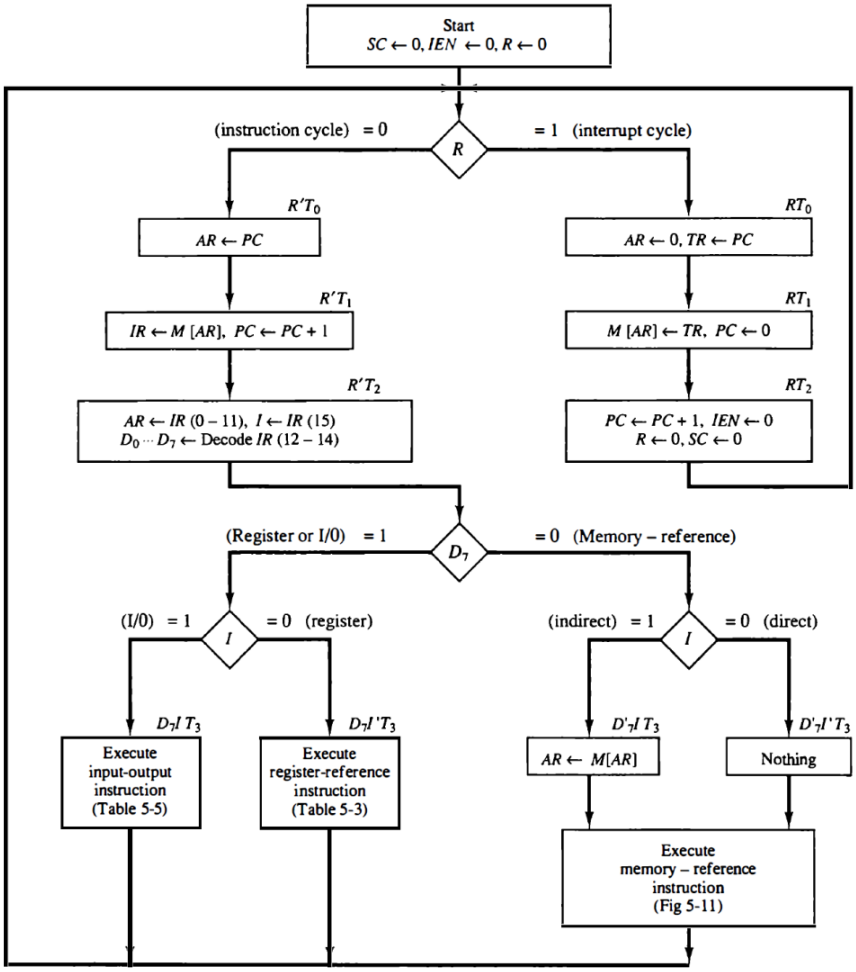



**Figure 5-15** Flowchart for computer operation.

Note how at the bottom of each column, it says "execute the instruction". We have a table with the exact instructions for each of these operations, or we can come up with it logically.

| Memory-reference: | | |
|---|---|---|
| AND | $D_0T_4$: | $DR \leftarrow M[AR]$ |
| | $D_0T_5$: | $AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$ |
| ADD | $D_1T_4$: | $DR \leftarrow M[AR]$ |
| | $D_1T_5$: | $AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$ |
| LDA | $D_2T_4$: | $DR \leftarrow M[AR]$ |
| | $D_2T_5$: | $AC \leftarrow DR$, $SC \leftarrow 0$ |
| STA | $D_3T_4$: | $M[AR] \leftarrow AC$, $SC \leftarrow 0$ |
| BUN | $D_4T_4$: | $PC \leftarrow AR$, $SC \leftarrow 0$ |
| BSA | $D_5T_4$: | $M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$ |
| | $D_5T_5$: | $PC \leftarrow AR$, $SC \leftarrow 0$ |
| ISZ | $D_6T_4$: | $DR \leftarrow M[AR]$ |
| | $D_6T_5$: | $DR \leftarrow DR + 1$ |
| | $D_6T_6$: | $M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$ |

We also have one for RRI and IOI.

### 5.6 Interrupt

The ION and IOF instructions will enable and disable the interrupt (IEN register) respectively.

When the interrupt is on (IEN=1), we use a **flag** called R (just a flip flop).

R is checked at the **start of each instruction cycle**. If it is 1, then we branch to M[0] and execute the **interrupt sequence** stored there.

## 6 Chapter 6

Machine language is a sequence of commands representing the exact content in memory.

Assembly is like machine language but there are comments, and variables.

### 6.1 Assembly Syntax

In assembly, we have 3 columns.

| Label | Instruction | Comment |
|---|---|---|
| Optional | Mandatory | Optional |

This is an example of assembly code with comments

```
     ORG 100 /sets origin of program (here) to 100
     LDA HEX A2 /loads hex value A2 into AC
CTR, DEC -2 /creates counter variable with decimal value -2
LOP, ISZ CTR /increments CTR, skips next line if 0 (will be -1)
     BUN LOP /branches to the line with LOP (the ISZ)
     HLT /halts the computer
```

This code will loop twice before getting 0 in CTR, then it will skip the BUN line, and HLT the computer.

The *label* can be a string up to 3 letters, that cannot start with a number.

The *instruction* can be any RRI, MRI, or IOI instruction, or a psudocode (ORG, END, DEC, HEX).

To use **indirect addressing**, we put an I at the end of the instruction.

```
ORG 100 /sets origin of program (here) to 100
LDA x I /loads M[M[x]]
x, HEX 111
END
```

### 6.2 Assembler

The assembler converts assembly to binary that the computer can handle.

The first pass generates the address symbol table which maps the variables to their binary equivalent.

The second pass translates everything else to binary.

## 6.3 Double Precision

When we want to add 2 numbers which have double precision (2 memory locations for each number), we need to be careful.

1. Add the lower halves together
2. Save result
3. Get the carry from E into AC (using CLA, then CIL)
4. Add both upper halves to the carry
5. Save result

## 6.4 Subroutines

A subroutine is called using the **BSA** instruction. We will BSA to the **address of the subroutine** (which *saves the current address at the address of subroutine*), and then at the end of the subroutine, BUN back to the main code (BUN to *start of subroutine* indirectly).

To pass data from the main to subroutine, we can either store it in a register, or/and store it after the BSA call.

```
    ORG 100
    BSA SUB /call subroutine
    HEX 123 /parameter passed into subroutine
    HLT /program done
SUB HEX 0 /start of subroutine, will store location to after BSA
    LDA SUB /since SUB now has HEX 123, loads that
    ISZ BSA /we do not want to return to 123, but to HLT
    BUN BSA /this returns to the HLT line
```