Systems and Devices 1
Lec 5c : The Computer

Before we get started ...

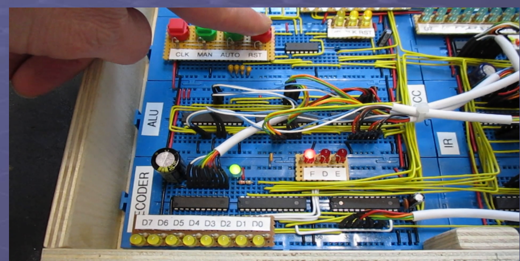
- We now have a “fully functioning” computer.
  - 12 instructions
    - MOVE, LOAD, STORE
    - ADD, SUB, ADDM, SUBM
    - Bitwise-AND
    - JUMP, JUMPZ, JUMPNZ
  - 3 addressing modes, 2 data types
    - Immediate, Absolute, Direct.
    - Signed, Unsigned 8-bit data types.
  - 256 x 16bit memory
    - 16-bit instructions, 8-bit variables
- What can we do with it? How can the computer interact with the real world?

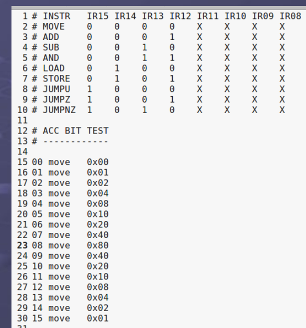
Instruction set

RTL	ENCODING	ASSEMBLER
Move KK : ACC <- KK	0000 XXXX KKKKKKKK	MOVE 0x01
Add KK : ACC <- ACC + KK	0001 XXXX KKKKKKKK	ADD 0x23
Sub KK : ACC <- ACC - KK	0010 XXXX KKKKKKKK	SUB 0x45
And KK : ACC <- ACC & KK	0011 XXXX KKKKKKKK	AND 0x67
Load AA : ACC <- M[AA]	0100 XXXX AAAAAAAA	LOAD 0x89
Store AA : M[AA] <- ACC	0101 XXXX AAAAAAAA	STORE 0x89
AddM AA : ACC <- ACC + M[AA]	0110 XXXX AAAAAAAA	ADDM 0xAB
SubM AA : ACC <- ACC - M[AA]	0111 XXXX AAAAAAAA	SUBM 0xAB
JumpU AA : PC <- AA	1000 XXXX AAAAAAAA	JUMPU 0xCD
JumpZ AA : IF Z=1 PC <- AA ELSE PC <- PC + 1	1001 XXXX AAAAAAAA	JUMPZ 0xEF
JumpNZ AA : IF Z=0 PC <- AA ELSE PC <- PC + 1	1010 XXXX AAAAAAAA	JUMPNZ 0xF0

- SimpleCPU machine-level instructions
  - Everything has to be implement from these instructions

Demo : System Test





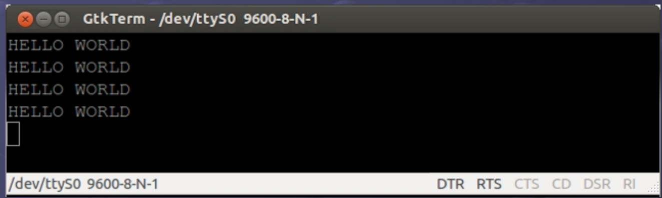
```
# INSTR   IR15 IR14 IR13 IR12 IR11 IR10 IR09 IR08
# MOVE    0    0    0    0    X    X    X    X
# ADD     0    0    0    1    X    X    X    X
# SUB     0    0    1    0    X    X    X    X
# AND     0    0    1    1    X    X    X    X
# LOAD    0    1    0    0    X    X    X    X
# STORE   0    1    0    1    X    X    X    X
# JUMPU   1    0    0    0    X    X    X    X
# JUMPZ   1    0    0    1    X    X    X    X
# JUMPNZ  1    0    1    0    X    X    X    X

# ACC BIT TEST
# ------------

00 move   0x00
01 move   0x01
02 move   0x02
03 move   0x04
04 move   0x08
05 move   0x10
06 move   0x20
07 move   0x40
08 move   0x80
09 move   0x40
10 move   0x20
11 move   0x10
12 move   0x08
13 move   0x04
14 move   0x02
15 move   0x01
```

- Before we can write our “first” program we need to test if the hardware is working correctly e.g. are there any damaged ICs or missing wires, …
- Therefore, our first program is a test program: test.asm
  - Lets go through the code …

Slide 5

Demo : Hello World

- Traditionally the first program you write on any new machine is one that prints "Hello World".
  - The FPGA board used to implement SimpleCPU does not have a display
  - Two choices:
    - LCD
    - Serial terminal



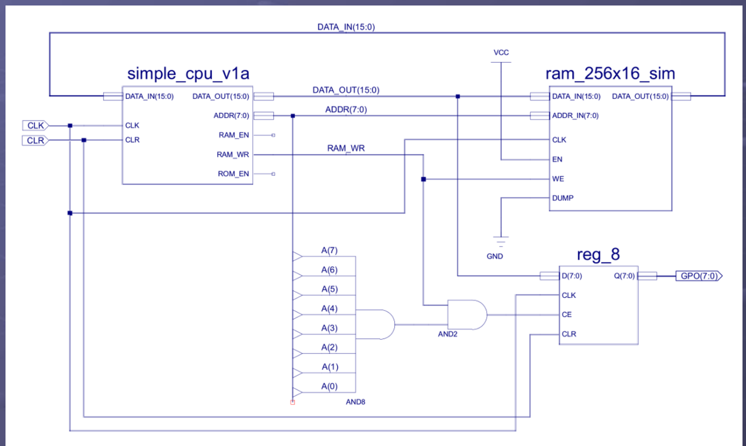
Slide 6

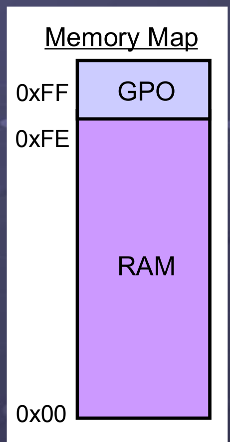
GPIO

- To interface the processor to the outside world we commonly use General Purpose Input Output pins.
  - Programmer controlled digital interface devices that can read inputs and control outputs in the real world.
  - Software controlled IO, no hardware support.
- Alternatively, application specific peripheral devices:
  - Parallel Port : data transferred using multiple wires e.g. comparable to a bus inside the processor, additional hardware support to synchronise data transfers, buffer data.
  - Serial Port : data transferred using a single wire i.e. one bit at a time, additional hardware support to convert parallel data to serial and vice versa, hardwired control logic, data buffers etc.

Slide 7

Parallel Port

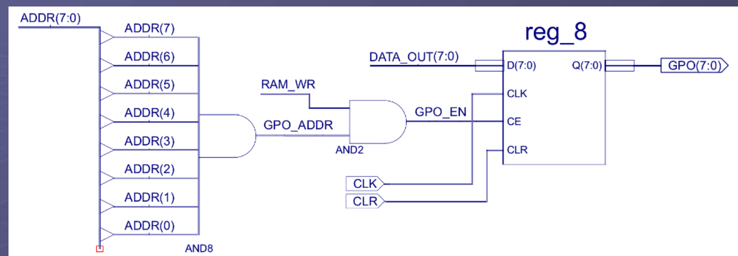




- Memory mapped (address 0xFF) output port
  - 8 bit register, Q outputs drive external signals connected to LCD display

Slide 8

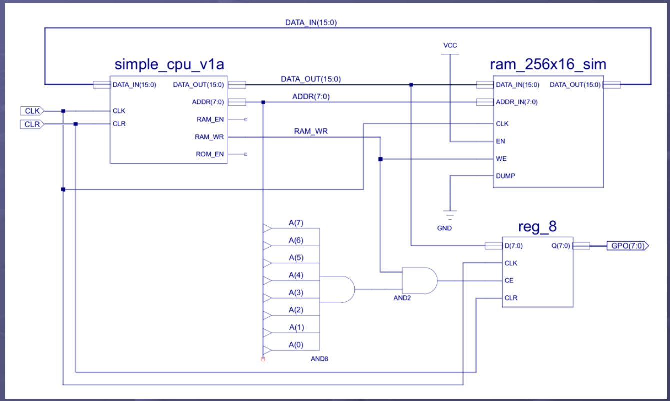
Parallel Port

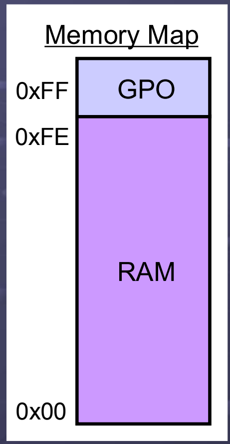


- Address decoding logic only enables output register when the processor writes to address 0xFF.
  - Q outputs updated with value on DATA_OUT bus (bits 7:0)

Slide 9

Parallel Port





- Quick Quizz (be careful trick question)
  - Can the processor read the output of the parallel port?
    - What happens when the CPU writes to ADDR 0xFF?
    - What happens when the CPU reads from ADDR 0xFF?

Slide 10

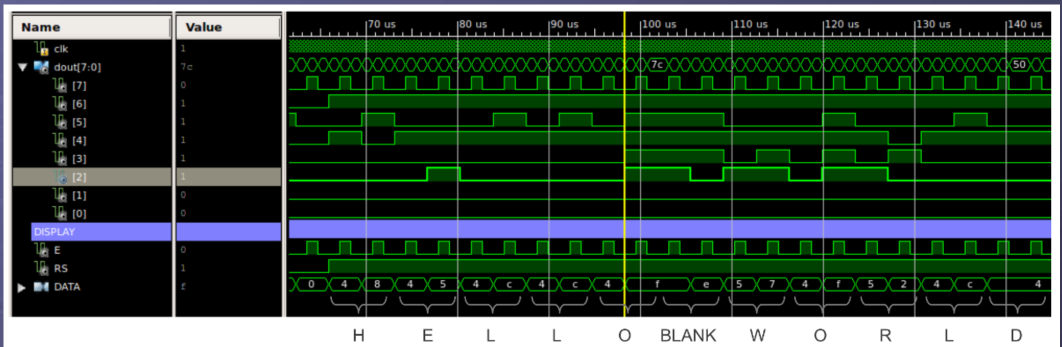
Text characters

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

- ASCII: 95 alpha-numeric, 33 control characters
  - Used in a later lab.

Slide 11

Parallel Port

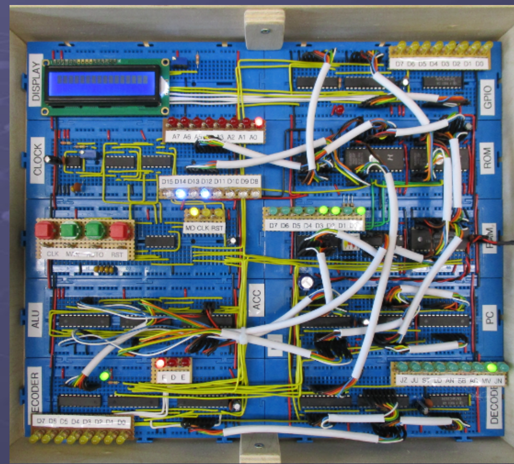


- LCD module is controlled using a 6 bit bus
  - E (7) : enable, active high, indicates that RS and DATA lines are valid and can be read.
  - RS (6) : register select, 0 = command, 1 = character data
  - Data (5:2) : 4 bit data bus, chars transferred as two nibbles.

Slide 12

Demo : Hello World

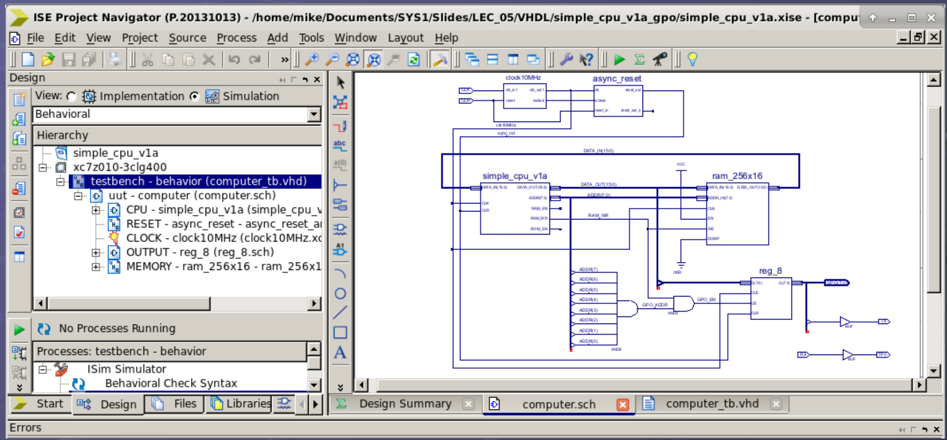
```
#
# INTERFACE
# ---------

# Q7;    /* E  */
# Q6;    /* RS */
# Q5;    /* D7 */
# Q4;    /* D6 */
# Q3;    /* D5 */
# Q2;    /* D4 */
# Q1;    /* NU */
# Q0;    /* NU */

# Initialise display
# ------------------

start:
    move   0x00   # load ACC with 0
    store  0xFF   # write to output port

# 0011 0011 Initialise
# --------------------

#         E RS D7 D6 | D5 D4 X X
# 0011 - 0 0  0  0  | 1  1  0 0  = 0x0C
# 0011 - 0 0  0  0  | 1  1  0 0  = 0x0C

    move   0x0C   # transfer 0011
    store  0xFF   # write to output port
    add    0x80   # set E high
    store  0xFF   # write to output port
    sub    0x80   # set E low
    store  0xFF   # write to output port
```



- Software defined parallel port
  - Bit flipping of control lines, bitwise operations etc.

Slide 13

Worked Example : SimpleCPU_PIO



- Parallel IO (PIO)
  - Run-time : approximately 700 us at 10MHz

Slide 14

Worked Example : SimpleCPU_PIO

```
# Initialise display
# -----------------

start:
  move   0x00   # load ACC with 0
  store  0xFF   # write to output port

# 0011 0011 Initialise
# -------------------

#           E RS D7 D6 | D5 D4 X X
# 0011 - 0 0  0  0  | 1  1  0 0  = 0x0C
# 0011 - 0 0  0  0  | 1  1  0 0  = 0x0C

  move   0x0C   # transfer 0011
  store  0xFF   # write to output port
  add    0x80   # set E high
  store  0xFF   # write to output port
  sub    0x80   # set E low
  store  0xFF   # write to output port

  move   0x0C   # transfer 0011
  store  0xFF   # write to output port
  add    0x80   # set E high
  store  0xFF   # write to output port
  sub    0x80   # set E low
  store  0xFF   # write to output port
```

```
# 0100 1000 Print 'H'
# -------------------

#           E RS D7 D6 | D5 D4 X X
# 0100 - 0 1  0  1  | 0  0  0 0  = 0x50
# 1000 - 0 1  1  0  | 0  0  0 0  = 0x60

  move   0x50   # transfer 0100
  store  0xFF   # write to output port
  add    0x80   # set E high
  store  0xFF   # write to output port
  sub    0x80   # set E low
  store  0xFF   # write to output port

  move   0x60   # transfer 1000
  store  0xFF   # write to output port
  add    0x80   # set E high
  store  0xFF   # write to output port
  sub    0x80   # set E low
  store  0xFF   # write to output port
```

- Lets go through the code …

Slide 15

Worked Example : SimpleCPU_PIO

```
define( lcd_write_nibble,`move $1
store  0xFF
add    0x80
store  0xFF
sub    0x80
store  0xFF')

define( lcd_write_command, `lcd_write_nibble(  eval( (`$1' & 240) >> 2) )
lcd_write_nibble( eval( (`$1' & 15) << 2 ))' )

define( lcd_write_data, `lcd_write_nibble(  eval( ((`$1' & 240) >> 2) | 64 ))
lcd_write_nibble( eval( ((`$1' & 15 ) << 2) | 64 ))' )
```

```
# 0011 0011 Initialise
# -------------------

#         E RS D7 D6 | D5 D4 X X
# 0011 - 0 0  0  0  | 1  1  0 0  = 0x0C
# 0011 - 0 0  0  0  | 1  1  0 0  = 0x0C

  lcd_write_command( 0x33 )
```

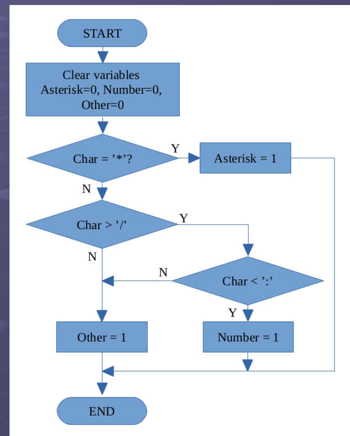
```
# 0100 1000 Print 'H'
# -------------------

#         E RS D7 D6 | D5 D4 X X
# 0100 - 0 1  0  1  | 0  0  0 0  = 0x50
# 1000 - 0 1  1  0  | 0  0  0 0  = 0x60

  lcd_write_data( 0x48 )
```

- Alternative implementation using MACROs
  - We will look at the M4 pre-processor in a later Lecture/Lab

Slide 16

Programming structures

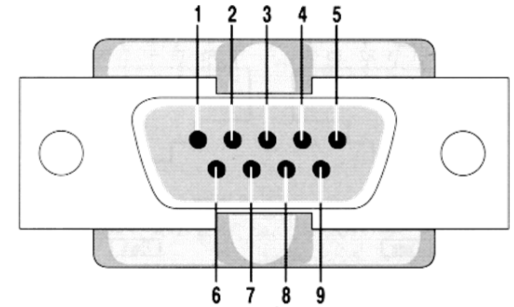
```
Asterisk = 0
Number = 0
Other = 0

IF Char = '*'
THEN
   Asterisk = 1
ELSIF Char > '/'
THEN
   IF Char < ':'
   THEN
      Number = 1
   ELSE
      Other = 1
   END IF
ELSE
   Other = 1
END IF
```




```
start:
   load Char
   sub 0x2A
   jumpnz test1
   move 1
   store Asterisk
   jumpu end

test1:
   sub 0x05
   and 0x80
   jumpz test2

setOther:
   move 1
   store Other
   jumpu end

test2:
   load Char
   sub 0x3A
   and 0x80
   jumpz setOther
   move 1
   store Number

end:
   jump end

Asterisk:
   .data 0
Number:
   .data 0
Other:
   .data 0
```

- Programs so far have had a single basic block of code, missing: selection and iteration (lab6).
  - Identify character, 0=False, 1=True.

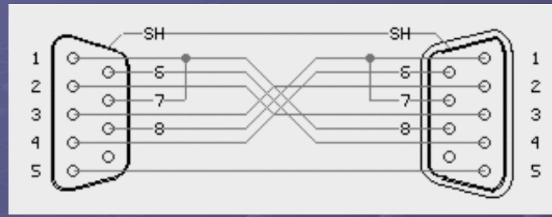
Serial Port

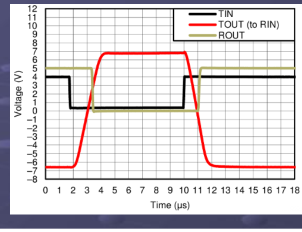


Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

- RS-232/V.24 pin out on a DB9 connector

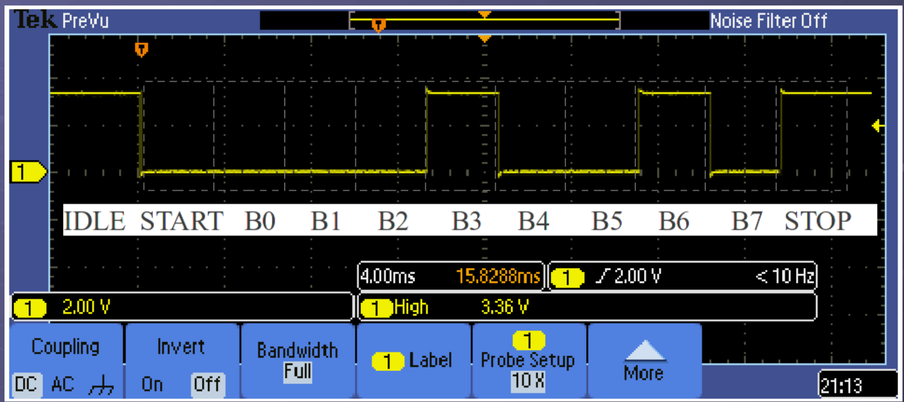
Serial Port





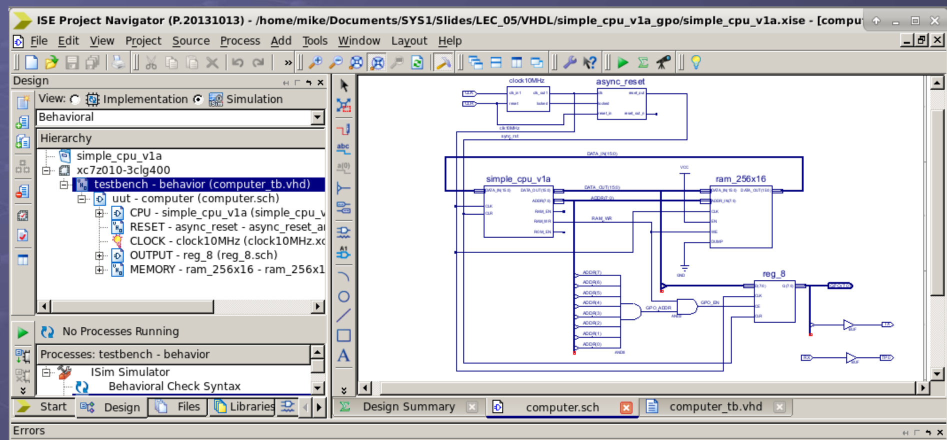
- RS232 : Point–to–point connection, 15M, 256Kbps
  - +3 to +12 volts indicates an "ON or 0-state (SPACE)
  - -3 to -12 volts indicates an "OFF" 1-state (MARK)
- Inverter drivers converting +12 / -12 voltages to logic 0 / 1
  - MAX232 driver / receiver

Serial Port



- Serial data link, two wire interface: RxD, TxD + GND
  - ASCII data converted into a serial data packet e.g. letter “H”
    - Packet divided into equal time slices, each bit allocated one slice.
    - Communications speed, bits per second (bps) e.g. 300bps = 3.3ms

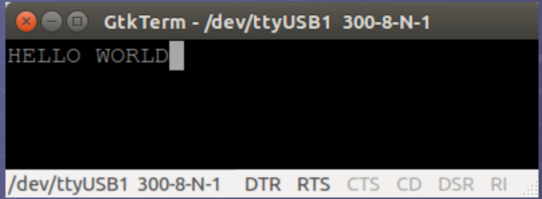
Worked Example : SimpleCPU_SIO



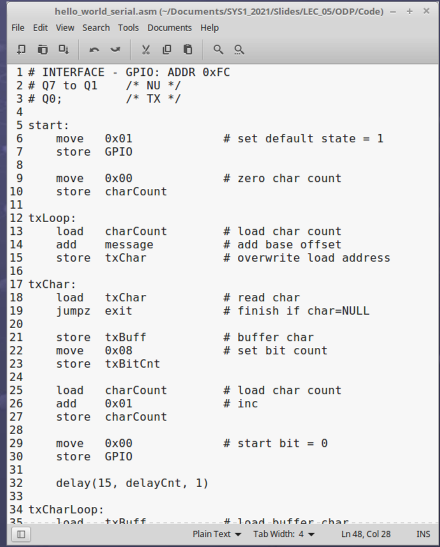
- Serial IO (SIO)
  - Run-time : approximately 300ms at 10MHz

Slide 21

Hello World



- Bit banging
  - "slang for various techniques for data transmission in which software is used to generate and process signals instead of dedicated hardware"
  - Processor running at 10MHz, therefore, bit-rate limited to a few 100 bps.



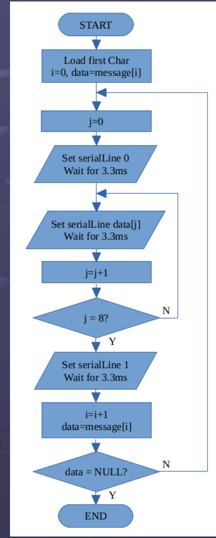

```
# INTERFACE - GPIO: ADDR 0xFC
# Q7 to Q1     /* NU */
# Q0;          /* TX */

start:
    move   0x01             # set default state = 1
    store  GPIO

    move   0x00             # zero char count
    store  charCount

txLoop:
    load   charCount        # load char count
    add    message          # add base offset
    store  txChar           # overwrite load address

txChar:
    load   txChar           # read char
    jumpz  exit             # finish if char=NULL

    store  txBuff           # buffer char
    move   0x08             # set bit count
    store  txBitCnt

    load   charCount        # load char count
    add    0x01             # inc
    store  charCount

    move   0x00             # start bit = 0
    store  GPIO

    delay(15, delayCnt, 1)

txCharLoop:
    load   txBuff           # load buffer char
```


Slide 22

Serial Port

- Pseudo code and flowchart
- Need to:
  - Select each BIT
  - Select each CHAR

```
i = 0
data = message[i]

while data != NULL:
   set serialLine low
   wait 3.3ms

   for j in range 0 to 7:
      set serialLine data[j]
      wait 3.3ms
   set serialLine high
   wait 3.3ms

   i = i + 1
   data = message[i]

i:
   .data 0
j:
   .data 0
data:
   .data 0

message:
   .data 'H','E','L','L','O',' '
   .data 'W','O','R','L','D','\0'
```



Slide 23

Select BIT

```
                 0100 1000   =     0x48 = 'H'

                                   TX START
                 0100 100|0|  =    TX 0   =   0x48
shift right      0010 010|0|  =    TX 0   =   0x24
shift right      0001 001|0|  =    TX 0   =   0x12
shift right      0000 100|1|  =    TX 1   =   0x09
shift right      0000 010|0|  =    TX 0   =   0x04
shift right      0000 001|0|  =    TX 0   =   0x02
shift right      0000 000|1|  =    TX 1   =   0x01
shift right      0000 000|0|  =    TX 0   =   0x00
                                   TX STOP
```

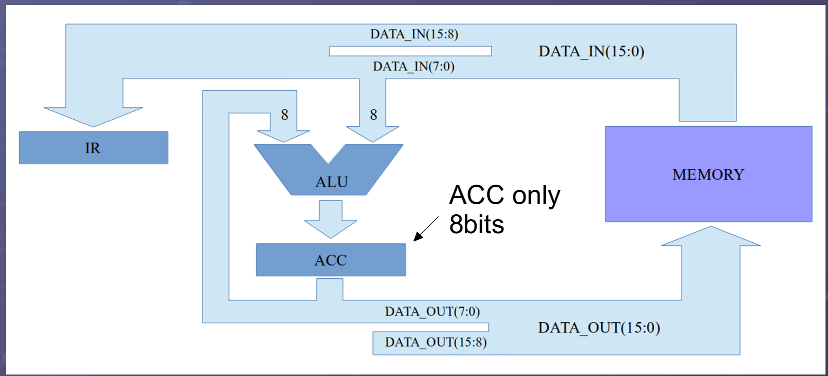
- Q: how can we shift ASCII data in the ACC right?

Slide 24

Serial Port

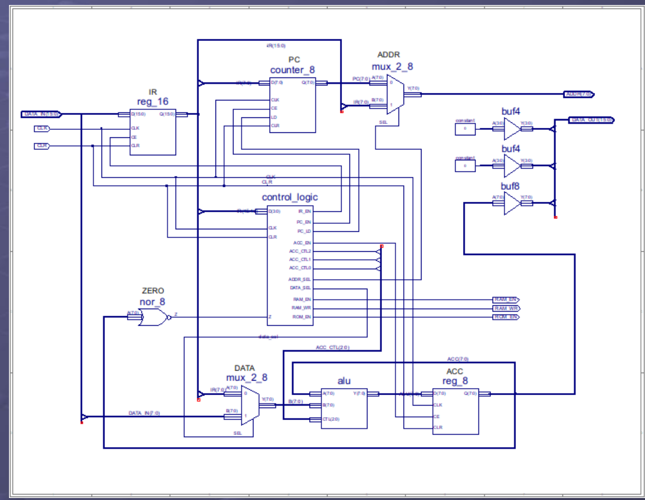
- A : write a program to divide the character data by 2 e.g. simple repeated subtraction.
  - Count how many times 2 can be subtracted without generating a carry.
- Q : how can we read character data from memory i.e. implement data = message[i]
- A : we can not i.e. at the moment we only have an absolute addressing mode LOAD instruction.
  - Read address can not be changed at runtime e.g. LOAD 55, we can not use a variable to address memory i.e. M[i].
  - However, we can bodge this by using self modifying code :)

Memory : Load / Store



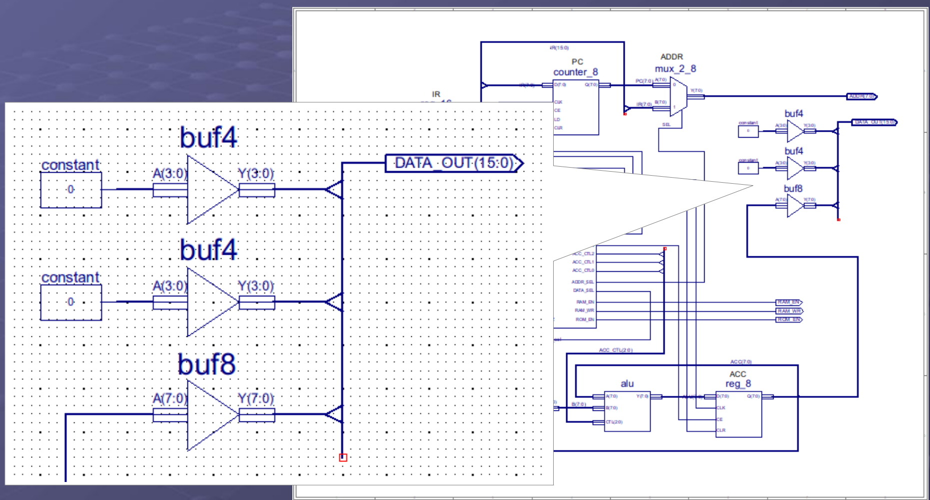
- For the simpleCPU_v1a we take the simple solution
  - Only read and write to lower 8-bits of a memory locations, downside wastes memory i.e. each time you declare a variable we will waste 8-bits.

SimpleCPU_v1a



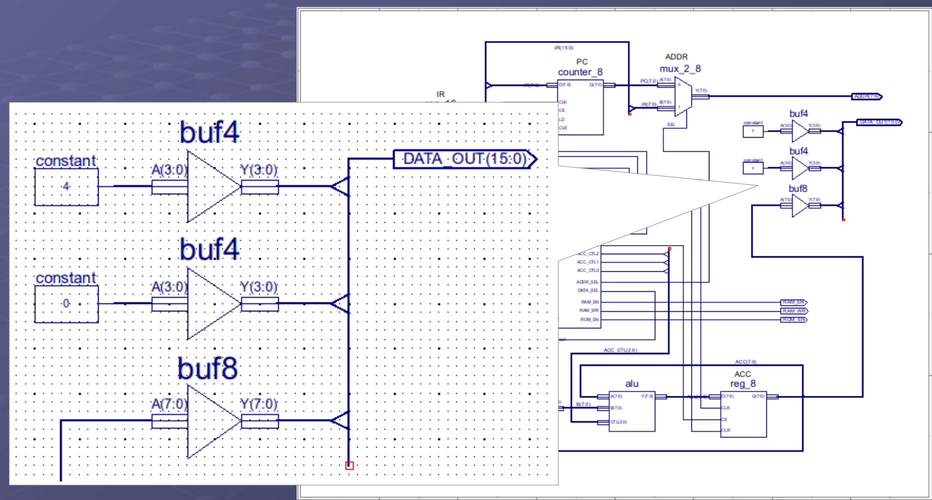
- Set the high byte of data_out = 0x00

SimpleCPU_v1a



- Set the high byte of data_out = 0x00

SimpleCPU_v1a



- Set the high byte of data_out = 0x40

Slide 29

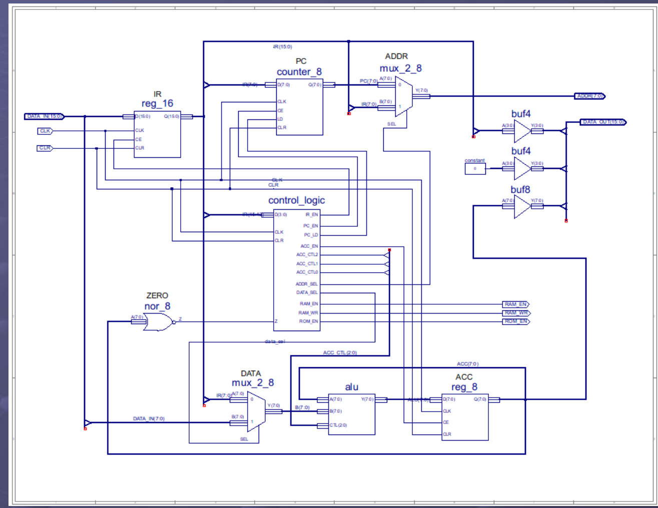
Quick Quizzzz

```
start:                   # ADDRESS   DATA
   load message          # 0         0x4007
   store char            # 1         0x5006
   load start            # 2         0x4000
   add 1                 # 3         0x1001
   store start           # 4         0x5000
   jump start            # 5         0x8000

char:
   .data 0               # 6         0x0000
message:
   .data 'H'             # 7         0x0048
   .data 'E'             # 8         0x0045
   .data 'L'             # 9         0x004C
   .data 'L'             # 10        0x004C
   .data 'O'             # 11        0x004F
```

- If we did hardwire the data_out bus to 0x40 || ACC, what does the above code do?

Slide 30

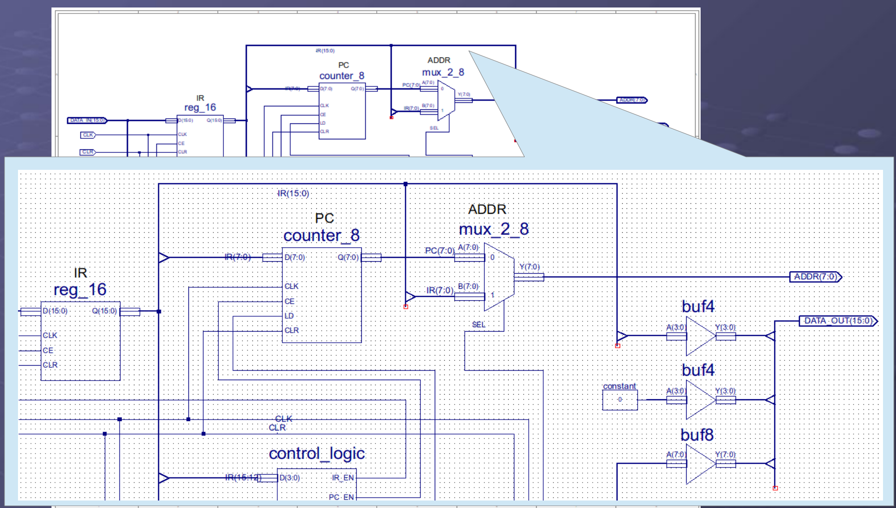
SimpleCPU_v1a



- Set the high byte of data_out = IR[11:8] || 0000

Slide 31

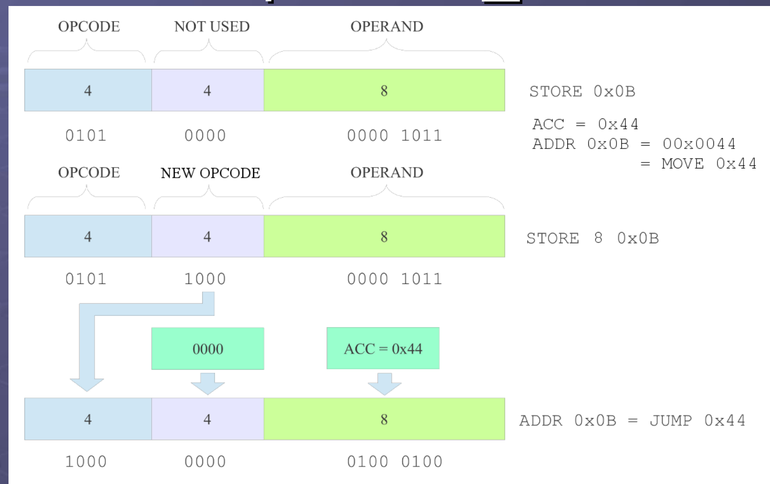
SimpleCPU_v1a



- Set the high byte of data_out = IR[11:8] || 0000

Slide 32

SimpleCPU_v1a



- Self-modifying code, what can go wrong :)
  - "New" 2-operand STORE instruction

Slide 33

A Wheeler JUMP

```
DATA = 22
X = FUNC(DATA)

FUNC:
    RETURN DATA×2
```

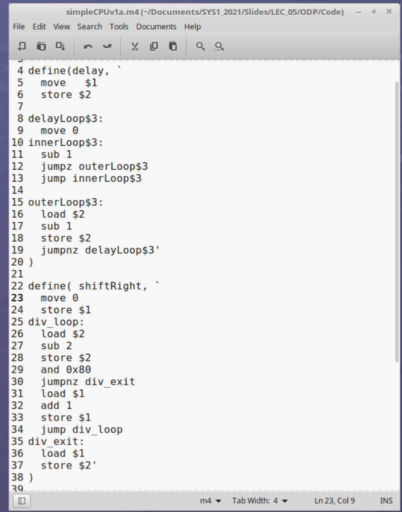
```
CODE:
    MOVE CODE
    JUMP FUNC
    ...

FUNC:
    ADD 2
    STORE 8 EXIT
    LOAD DATA
    ADDM DATA
EXIT:
    JUMP EXIT

DATA:
    22
```

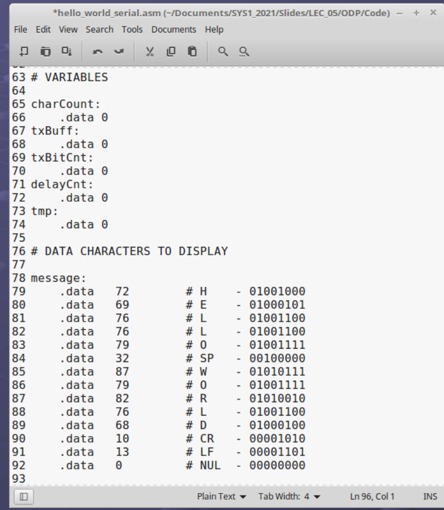
```
10 MOVE 10
11 JUMP 50
12 ...


50 ADD 2
51 STORE 8 54
52 LOAD 55
53 ADDM 55

54 JUMP 54

55 22
```

- The first implementation of a function call.
  - Quick Quizz : how does this code work?

Slide 34

Worked Example : SimpleCPU_SIO

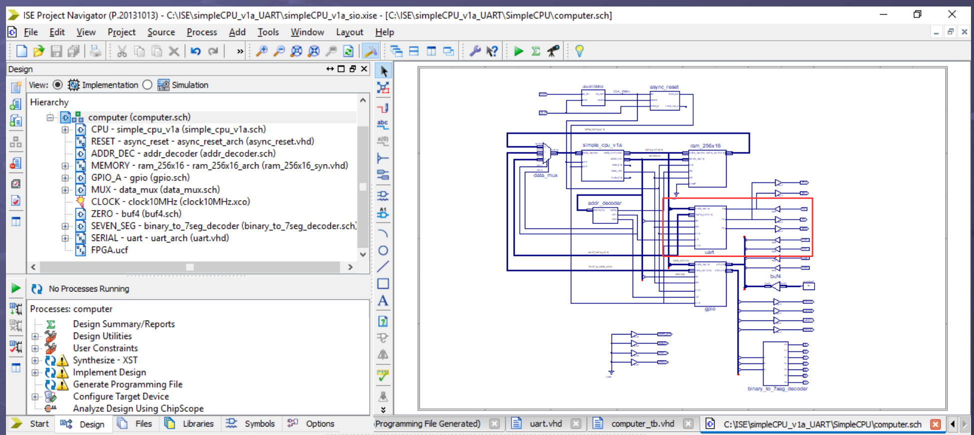




- Lets go through the code ...

Slide 35

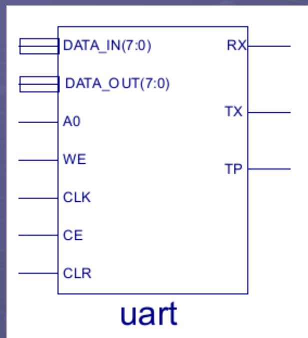
Worked Example : UART



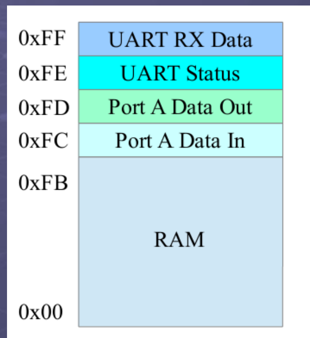
- Universal Asynchronous Receiver Transmitter unit
  - A hardware implemented serial port

Slide 36

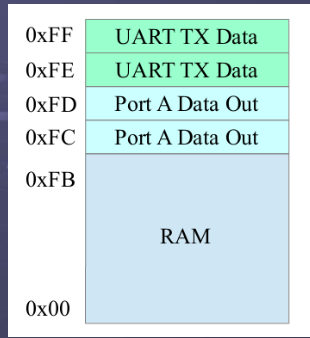
Worked Example : UART



SYMBOL



READ



WRITE

- Three memory mapped registers
  - TX data : write only, triggers automatic TX of value
  - RX data : read only, return received 8bit value (ASCII char)
  - Status : read only, return status of RX, TX and Buffer.

Worked Example : UART

```
# MEMORY MAP
# 0xFF - WR - UART tx
# 0xFF - RD - UART rx
# 0xFE - WR - UART tx
# 0xFE - RD - UART status

# 0xFD - WR - GPIO output
# 0xFD - RD - GPIO input
# 0xFC - WR - GPIO output
# 0xFC - RD - GPIO input

# UART REGISTERS
# TX : B7 - B0 data
# RX : B7 - B0 data

# STATUS REGISTER
# B7 : NU
# B6 : NU
# B5 : NU
# B4 : NU
# B3 : NU
# B2 : TX Idle
# B1 : RX Idle
# B0 : RX Valid

start:
    move 0x2A
    store 0xFF

wait1:
    load 0xFE   # wait for TX
    and 0x04
    jumpz wait1

loop:
    load 0xFE   # wait for RX
    and 0x01
    jumpz loop

    load 0xFF   # read RX
    store 0xFF  # wait RX to TX
    store 0xFD

wait2:
    load 0xFE   # wait for TX
    and 0x04
    jumpz wait2

    jump start
```

- Lets go through the code …
  - A lot simpler when its all done in hardware :)

Summary

- Key concepts
  - Control logic
    - Representing processor state
    - Generating control signals
  - Character (text) data types : ASCII
  - Parallel and Serial ports (IO)
    - Memory maps and memory mapped devices
  - Assembly language programming
    - Three case studies:
      - Easy : multiply 10 by 3
      - Medium : Hello World LCD
      - Hard : Hello World Serial (covered again in lab)