Spiral 3-1

Hardware/Software Interfacing

Learning Outcomes

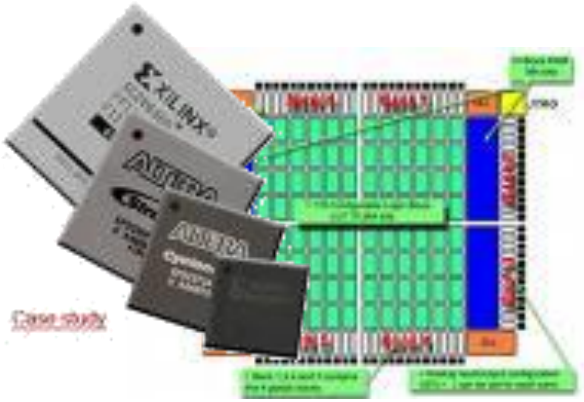
- I understand the PicoBlaze bus interface signals: PORT_ID, IN_PORT, OUT_PORT, WRITE_STROBE
- I understand how a memory map provides the agreement between addresses the software will use and that the hardware must recognize and respond to
- I understand how to build address decoding logic to ensure only the appropriate value/register is selected for a given PORTID
- For output, I can take a memory map and the PORTID and OUTDATA bits such that the appropriate data is input or saved in a register when an OUTPUT instruction is executed
- For input, I can take a memory map and the appropriate PORTID bits to build logic and muxes such that the appropriate data value is present at INDATA when an INPUT instruction is executed

FPGAS

Digital Design Targets

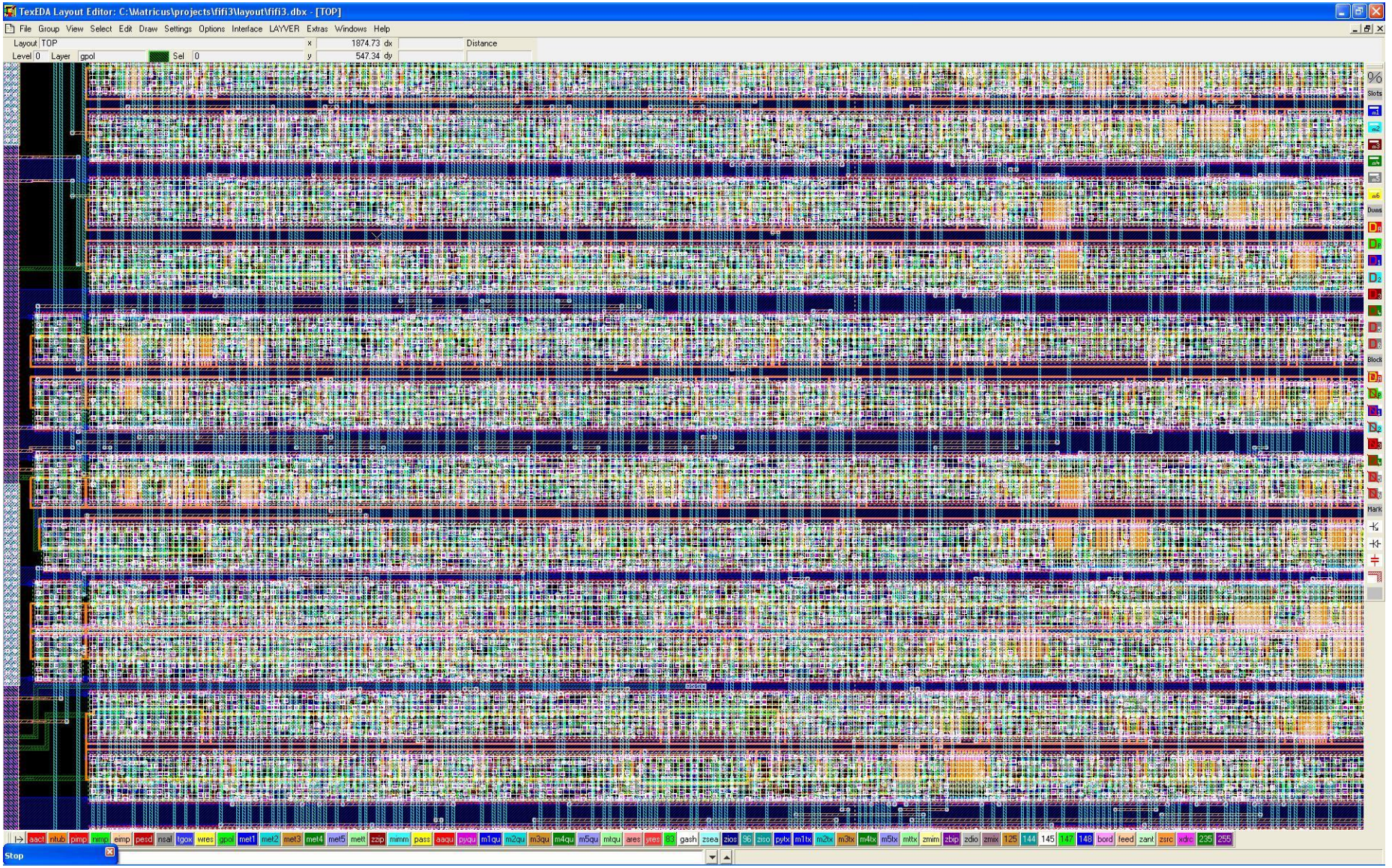
- Two possible implementation targets
  - Custom Chips (ASIC's = Application Specific Integrated Circuits): Physical gates are created on silicon to implement 1 particular design
  - FPGA (Field Programmable Gate Array's): "Programmable logic" using programmable memories to implement logic functions along with other logic resources tiled on the chip. Can implement any design and then be changed to implement a new one

In an ASIC design, a unique chip will be manufactured that implements our design at which point the HW design is fixed & cannot be changed (example: Pentium, etc.)



FPGA's have "logic resources" on them that we can configure to implement our specific design. We can then reconfigure it to implement another design

ASICs


TexEDA Layout Editor: C:\Matricus\projects\fifi3\layout\fifi3.dbx - [TOP]
File Group View Select Edit Draw Settings Options Interface LAYVER Extras Windows Help
Layout TOP
Level 0 Layer gpol
Sel 0
1874.73 dx
547.34 dy
Distance
Stop

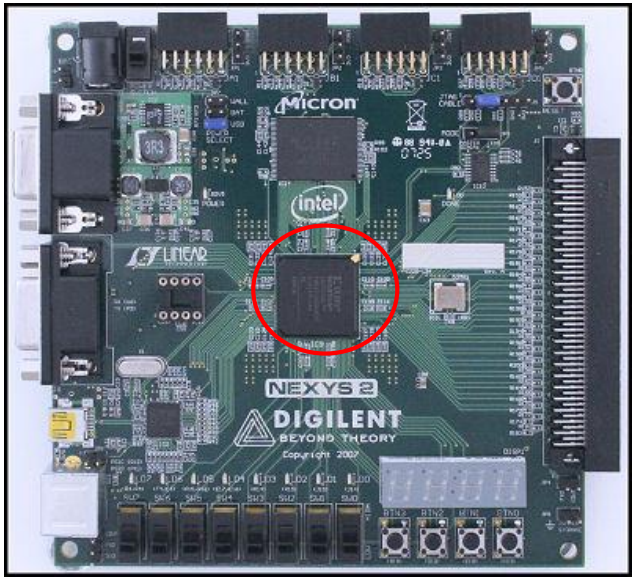
Implementation

- ASIC's
  - Use the CAD tools to synthesize and route a "netlist"
    - **Synthesis** = Takes logic description or logic schematic & converts to transistor level gates
    - **Place and Route** = Figure out where each gate should go on the chip)
  - Final "netlist" is sent to chip maker for production
  - Fabrication is very expensive (> $1 million) so get your design right the first time.
- FPGA's
  - Synthesis converts logic description to necessary LUT contents, etc.
  - Place and route produces a configuration for the FPGA chip
  - Can reconfigure FPGA as much as you like, so less important to get it right 1st time

ASIC's vs. FPGA's

- ASIC's
  - Faster
  - Handles Larger Designs
  - More Expensive
  - Less Flexible (Cannot be reconfigured to perform a new hardware function)
- FPGA's
  - Slower (extra logic to make it reconfigurable)
  - Smaller Designs
  - Less Expensive
  - Extremely Flexible

Xilinx Spartan 3E



Digilent Nexys-2 Board

- Has a Xilinx Spartan 3E FPGA (XC3S500e)
- 500K gate equivalent
- 9312 D-FF's on-board

On-board I/O

- (4) 7-Segment Displays
- (8) LED's
- (4) Push Buttons
- (8) Switches

Latest FPGA's

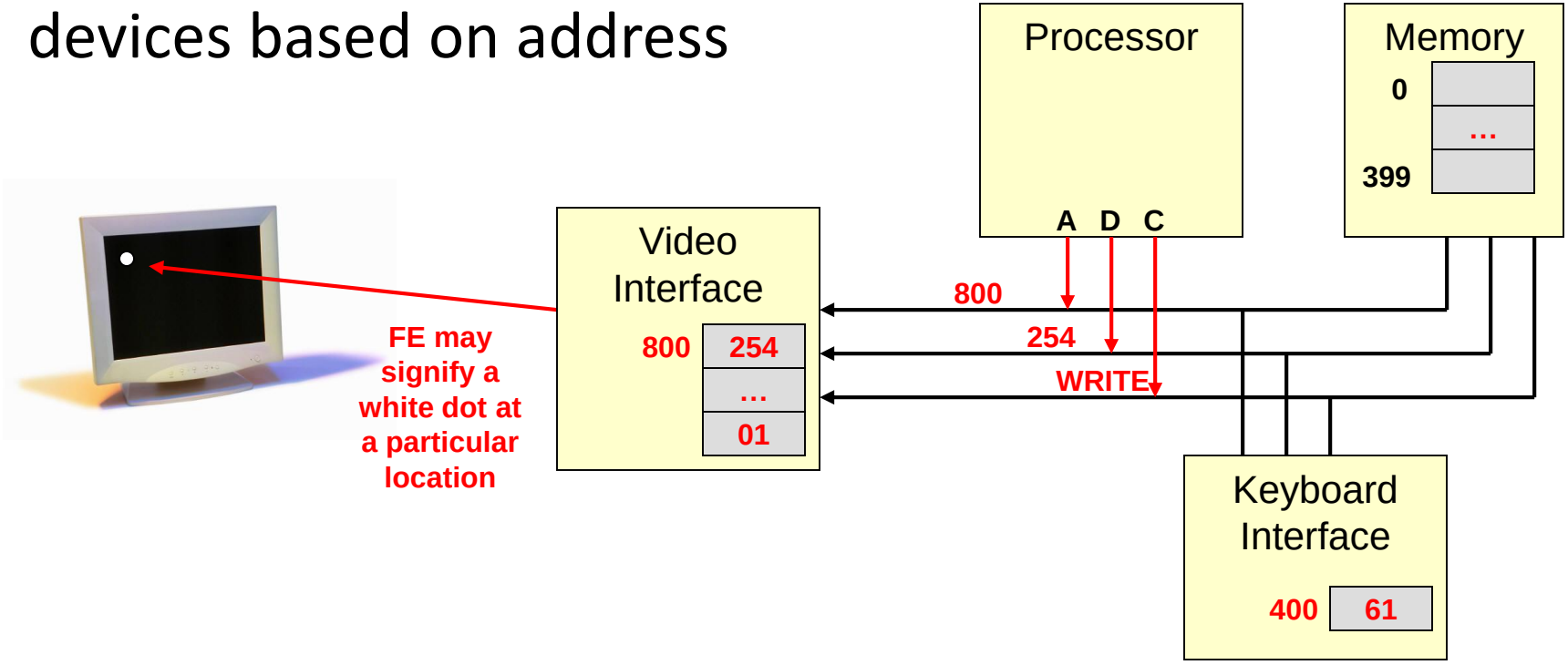
- SoC design (Xilinx Kintex [KU115])
  - Quad-Core ARM cores
  - DDR3 SDRAM Memory Interface
  - ~800 I/O Pins
  - Equiv. ~15M gate equivalent FPGA fabric
    - ~1M D-FFs + 552K LUTs
    - 1968 dedicated DSP "slices" 18x18 multiply + adder
    - 34.6 Megabits of onboard Block RAMs

Hardware/Software Interfacing

PICOBLAZE

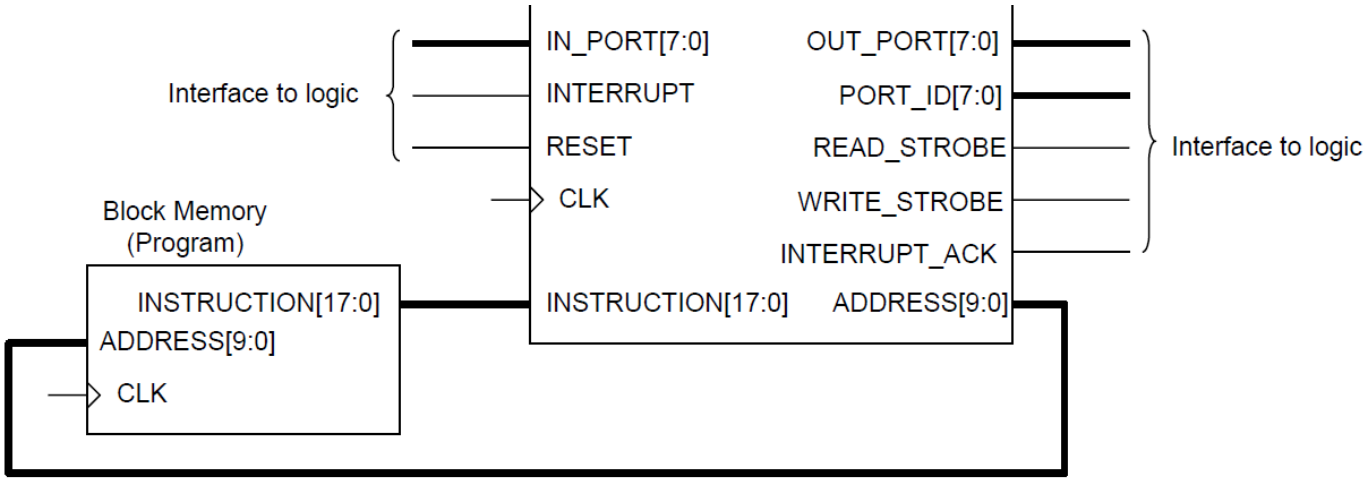
Input / Output

- The processor connects to peripherals and other logic via the bus (address, data, and control)
- Software running on the processor performs loads and stores that read and write data to and from these devices based on address



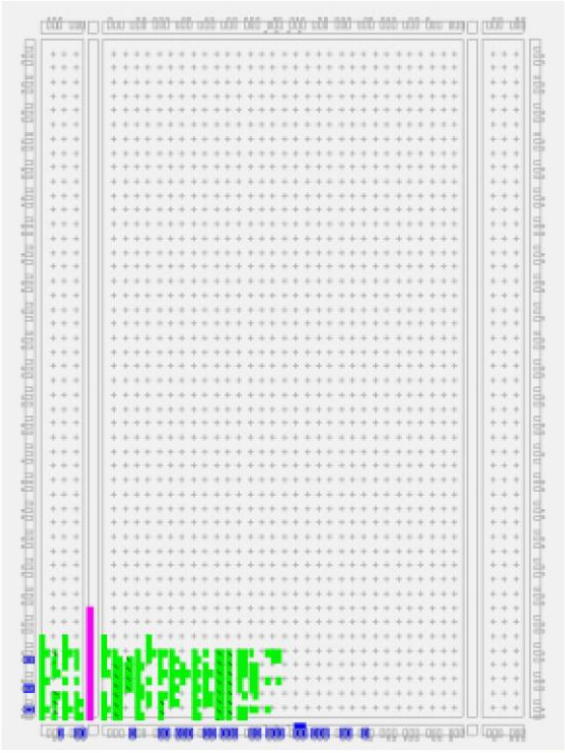
Introduction

- Picoblaze (aka KCPSM3) is an 8-bit soft-processor
  - The processor is not implemented directly in hardware on the FPGA but instead is just a description that is then synthesized using the same process as any of our other designs
  - It provides a bus interface that can be connected to custom logic that you design and then used to control that custom logic via software executing on the processor



KCPSM3 is small!

This plot from the Xilinx Floorplanner shows the same implementation of KCPSM3 in an XC3S200 Spartan-3 device. This makes it easier to appreciate the actual logic resources required by the macro without the interconnect obscuring the detail.

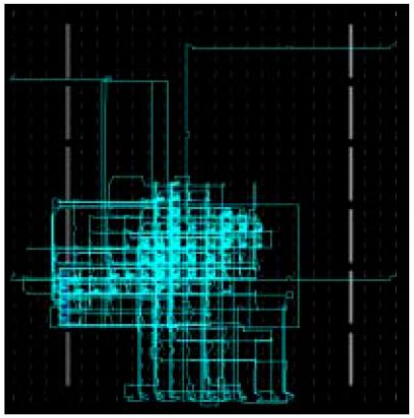


The placement in this Floorplanner view was achieved using a simple area constraint in the project UCF file.

```
INST processor_* LOC=SLICE_X0Y0:SLICE_X19Y4;
```

Such constraints are not required in normal designs and it has only been used in this case because so little of the device is occupied. Experiments have shown that placement constraints have very little effect on performance.

The FPGA Editor view shown to the right was the result when no constraints were used. The size is still 96 slices but this is now a little less obvious! The performance was actually a little higher than when using the area constraint indicating that a 'tidy' design is not always the fastest!



Taken from the KCPSM3 Manual

Size and Performance

The following device resource information is taken from the ISE reports for the KCPSM3 macro in an XC3S200 device. The reports reveal the features that are utilised and the efficiency of the macro. The 96 'slices' reported by the MAP process in this case may reduce to the minimum of 89 'slices' when greater packing is used to fit a complete design into a device.

XST Report

```
LUT1          : 2
LUT2          : 6
LUT3          : 68
LUT4          : 33
```
109 LUTs (55 slices)

```
MUXCY         : 39
MUXF5         : 9
XORCY         : 35
```
Carry and MUX logic (Free with LUTs)

```
FD            : 24
FDE           : 2
FDR           : 30
FDRE          : 8
FDRSE         : 10
FDS           : 2
```
76 Flip_flops (Free with LUTs)

```
RAM16X1D      : 8
RAM32X1S      : 10
RAM64X1S      : 8
```
Register bank (8 slices)
Call/Return Stack (10 slices)
Scratch Pad Memory (16 slices)

Total = 89 Slices

MAP Report

```
Number of occupied Slices : 96 out of 1920   5%
Number of Block RAMs      :  1 out of   12   8%
Total equivalent gate count for design:  74,814
```

12 × KCPSM3 can fit into the XC3S200 device (40% of the logic slices remaining). An equivalent gate count of 897,768 gates in a 200,000 gate device!

TRACE Report

```
Device,speed: xc3s200,-4 (PREVIEW 1.22 2003-03-16)
Minimum period:  11.403ns
(Maximum frequency:  87.696MHz)
```

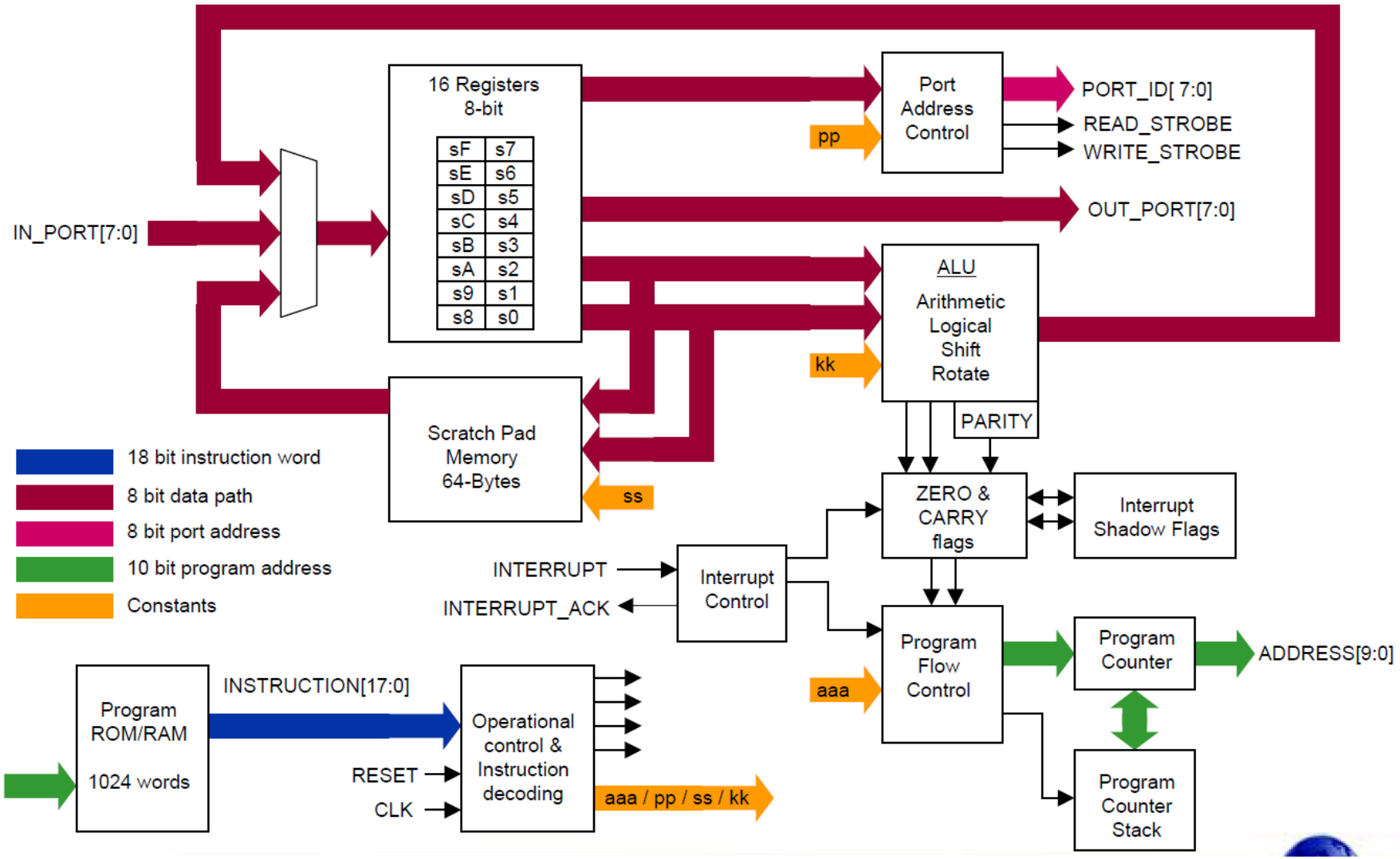
43.8 MIPS

TRACE Report for Virtex-IIPRO

```
Device,speed: xcvp2,-7 (ADVANCED 1.76 2003-03-16)
Minimum period:  7.505ns
(Maximum frequency:  133.245MHz)
```
66.6 MIPS

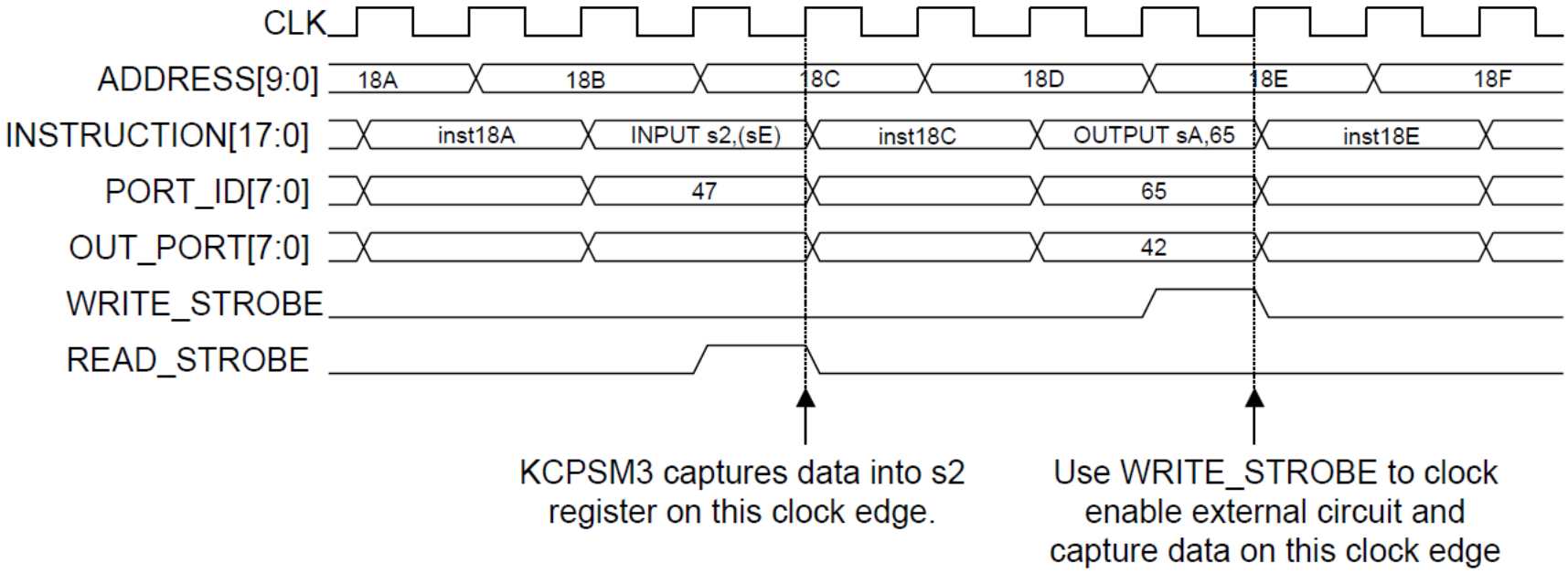
Taken from the KCPSM3 Manual

KCPSM3 Architecture



Taken from the KCPSM3 Manual

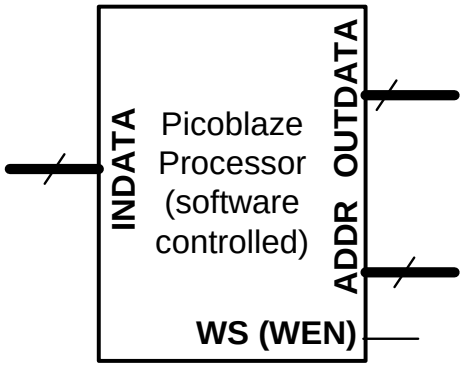
Input / Output Operations

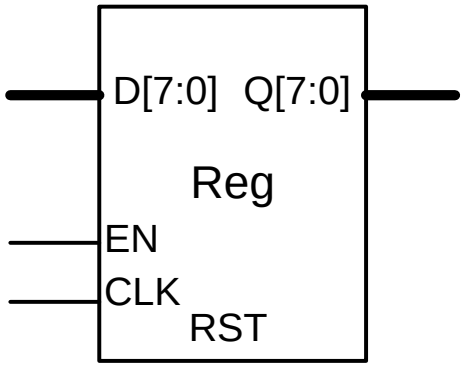


Taken from the KCPSM3 Manual

Exercise 1

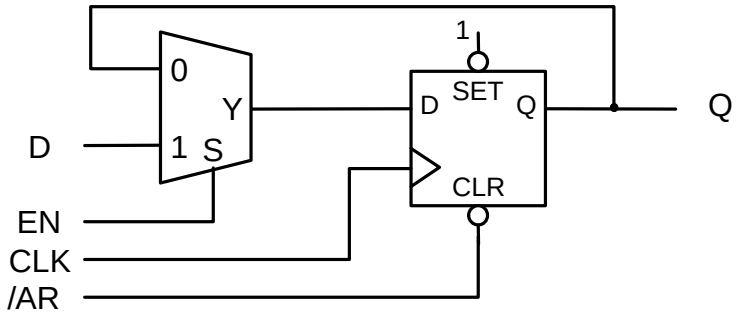
- Make the register below capture data (**`out_data`**) from your Picoblaze whenever it outputs address FF hex on (**`address`** or **`port_id`**)





Remember: Registers w/ Enables

- Registers (D-FF's) will sample the D bit every clock edge and pass it to Q
- Sometimes we may want to hold the value of Q and ignore D even at a clock edge
- We can add an enable input and some logic in front of the D-FF to accomplish this

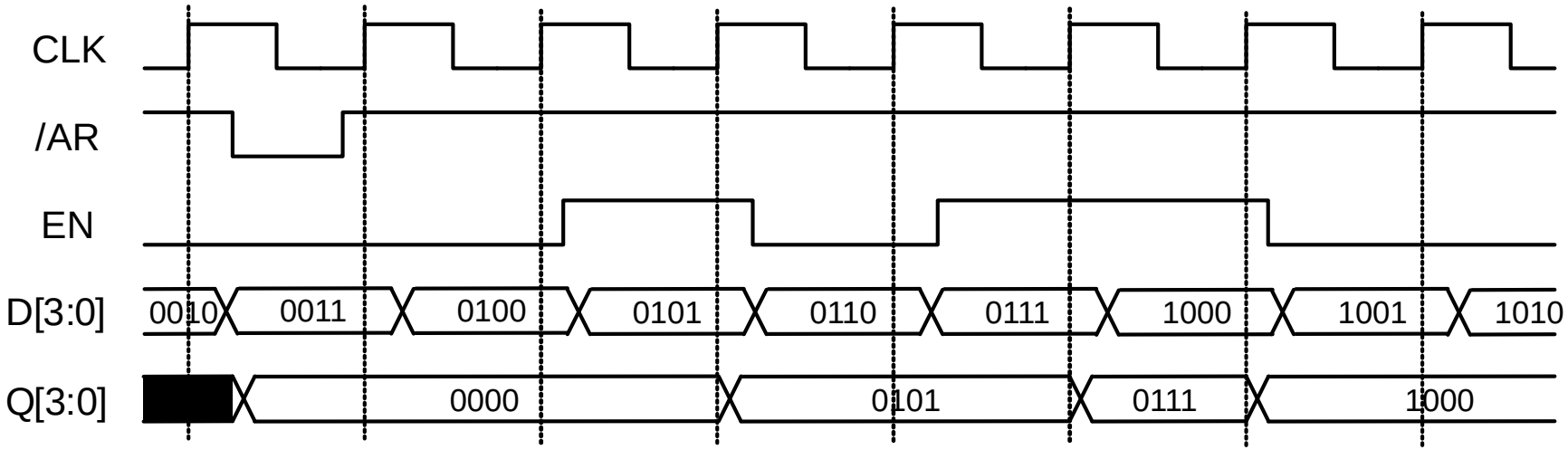


FF with Data Enable

CLK	/AR	EN	D_i	Q_i*
X	0	X	X	0
0,1	1	X	X	Q_i
↑↑	1	0	X	Q_i
↑↑	1	1	0	0
↑↑	1	1	1	1

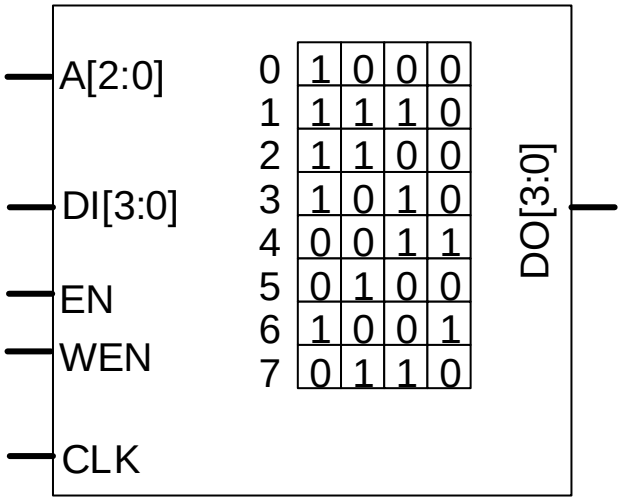
Registers w/ Enables

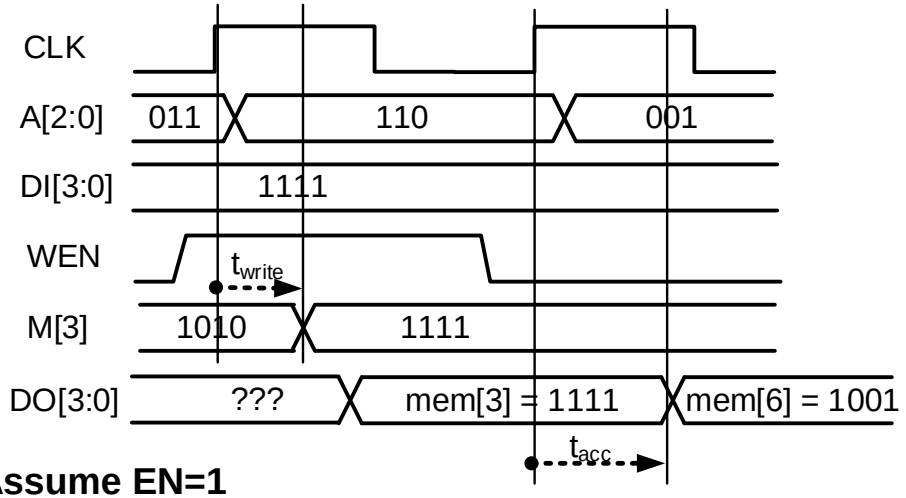
- The D value is sampled at the clock edge only if the enable is active
- Otherwise the current Q value is maintained



Recall Memory Interfaces

- We provide address and data
- EN = Overall enable (unless it is 1) the memory won't read or write (we assume EN=1)
- WEN = Write enable
  - 1 = Write / 0 = read

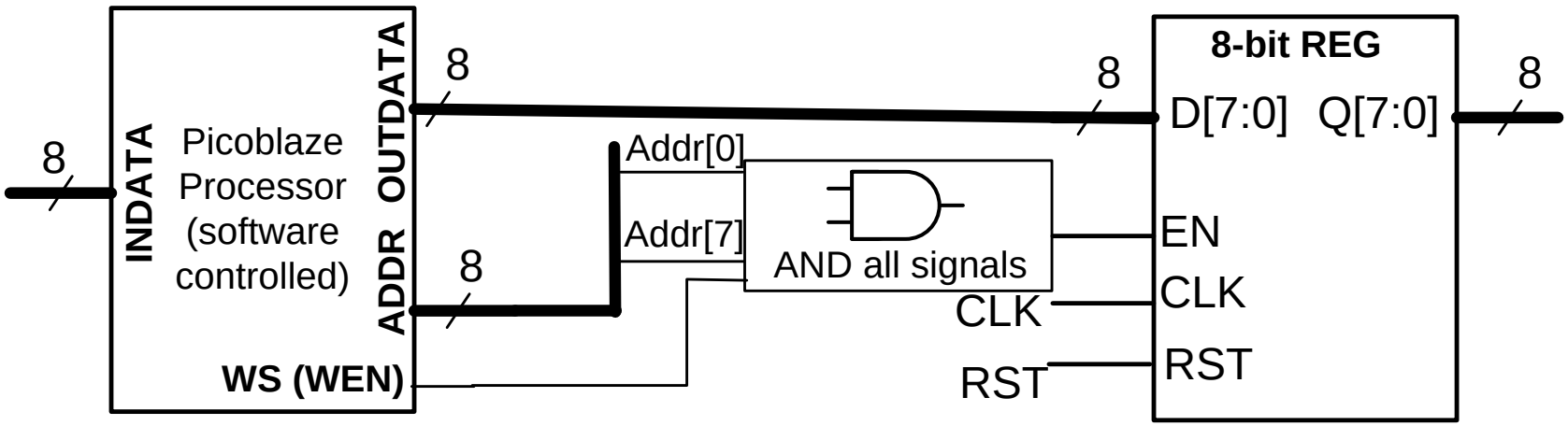




Assume EN=1

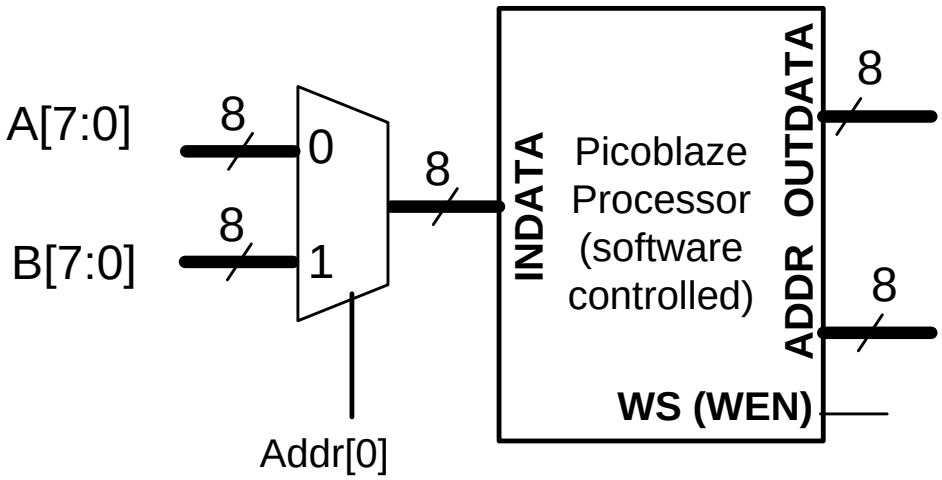
Exercise 1

- Make the register below capture data (**`out_data`**) from your Picoblaze whenever it outputs address FF hex on (**`address`** or **`port_id`**)



Exercise 2

- Use your PicoBlaze to receive input from A given address 00 hex and B for address 0x01 hex



Memory Maps

- A memory map shows what devices are assigned to a given address or address range that can then be accessed by the processor (and its software programs)

Dec	A7	A6	A5	A4	A3	A2	A1	A0	Assigned Device
00	0	0	0	0	0	0	0	0	Input Switches
01	0	0	0	0	0	0	0	1	open
02	0	0	0	0	0	0	1	0	open
…									
253	1	1	1	1	1	1	0	1	LEDs
254	1	1	1	1	1	1	1	0	open
255	1	1	1	1	1	1	1	1	7-Seg Display

Memory Maps

- Given an 8-bit address space (256 locations) and 3 devices that we want to interface to our microprocessor, we first must create the memory map
  - A 64 bytes (64x8) memory
  - A single 8-bit register
  - A single 1-bit D-FF

Dec	A7	A6	A5	A4	A3	A2	A1	A0	Assigned Device
00	0	0	0	0	0	0	0	0	64x8
01	0	0	0	0	0	0	0	1	Memory
…									
63	0	0	1	1	1	1	1	1	
64	0	1	0	0	0	0	0	0	8-bit Register
…									open
128	1	0	0	0	0	0	0	0	1-bit D-FF
…									open

Memory Maps

- Exercise: What is a minimal set of bits that could be used to distinguish each device from the others?
  - A 64 bytes (64x8) memory => A7-A6 = 0,0
  - A single 8-bit register => A7-A6 = 0,1
  - A single 1-bit D-FF => A7 = 1

Dec	A7	A6	A5	A4	A3	A2	A1	A0	Assigned Device
00	0	0	0	0	0	0	0	0	64x8
01	0	0	0	0	0	0	0	1	Memory
…									
63	0	0	1	1	1	1	1	1	
64	0	1	0	0	0	0	0	0	8-bit Register
…									open
128	1	0	0	0	0	0	0	0	1-bit D-FF
…									open

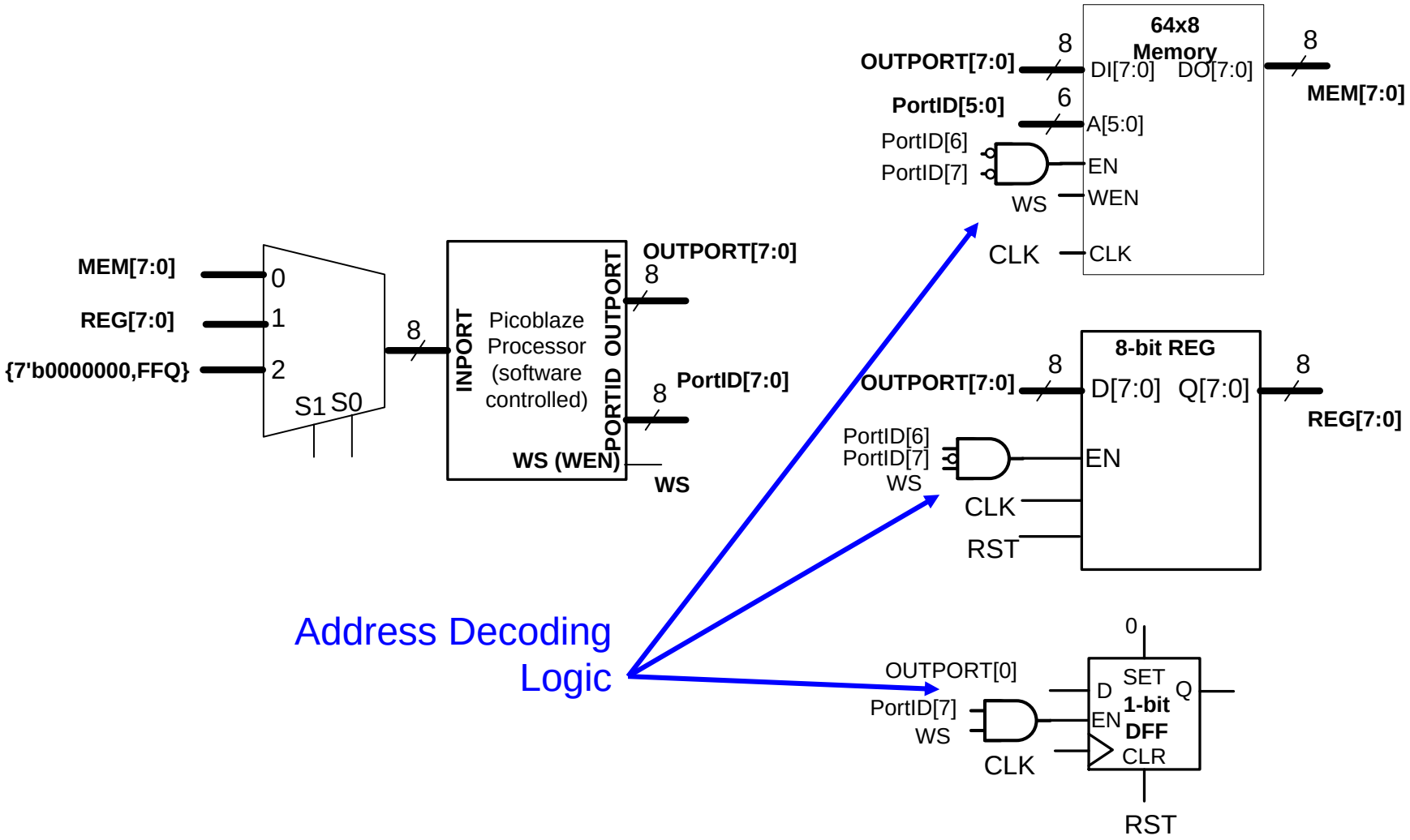
Memory Aliasing

- Given
  - A 64 bytes (64x8) memory => A7-A6 = 0,0
  - A single 8-bit register => A7-A6 = 0,1
  - A single 1-bit D-FF => A7 = 1
- By using don't care situations the 8-bit register will respond to any address where A7-A6 = 0,1 (i.e. 64-127) and similarly the 1-bit D-FF will respond to any address where A7=1 (i.e. 128-255)

Dec	A7	A6	A5	A4	A3	A2	A1	A0	Assigned Device
00	0	0	0	0	0	0	0	0	64x8
01	0	0	0	0	0	0	0	1	Memory
…									
63	0	0	1	1	1	1	1	1	
64	0	1	0	0	0	0	0	0	8-bit Register
…									~~open~~
128	1	0	0	0	0	0	0	0	1-bit D-FF
…									~~open~~

Address Decoding

- Address decoding refers to the process of enabling the correct device based on a specific address combination

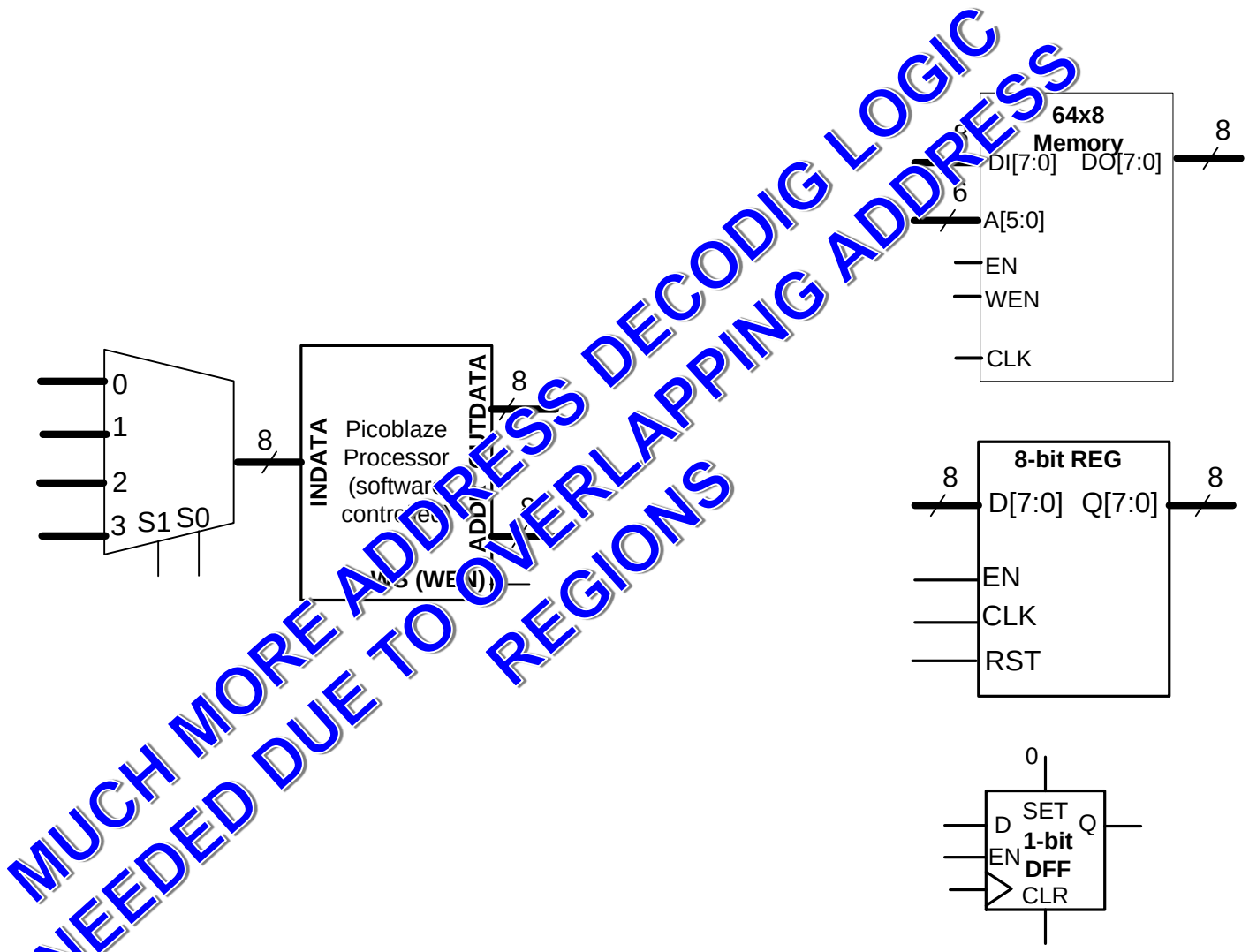


Memory Maps

- Exercise: Repeat the exercise to find a minimal set of bits that could be used to distinguish each device from the others?
  - A 64 bytes (64x8) memory => A6 + A6'(A5+A4+A3+A2+A1)
  - A single 8-bit register => A6'A5'A4'A3'A2'A1'A0'
  - A single 1-bit D-FF => A6'A5'A4'A3'A2'A1'A0

Dec	A7	A6	A5	A4	A3	A2	A1	A0	Assigned Device
00	0	0	0	0	0	0	0	0	8-bit Register
01	0	0	0	0	0	0	0	1	1-bit D-FF
02	0	0	0	0	0	0	1	0	64x8
03	0	0	0	0	0	0	1	1	Memory
04	0	0	0	0	0	1	0	0	
…									
64	0	1	0	0	0	0	0	0	
65	0	1	0	0	0	0	0	1	
66	0	1	0	0	0	0	1	0	open
…									open

Address Decoding Exercise 2



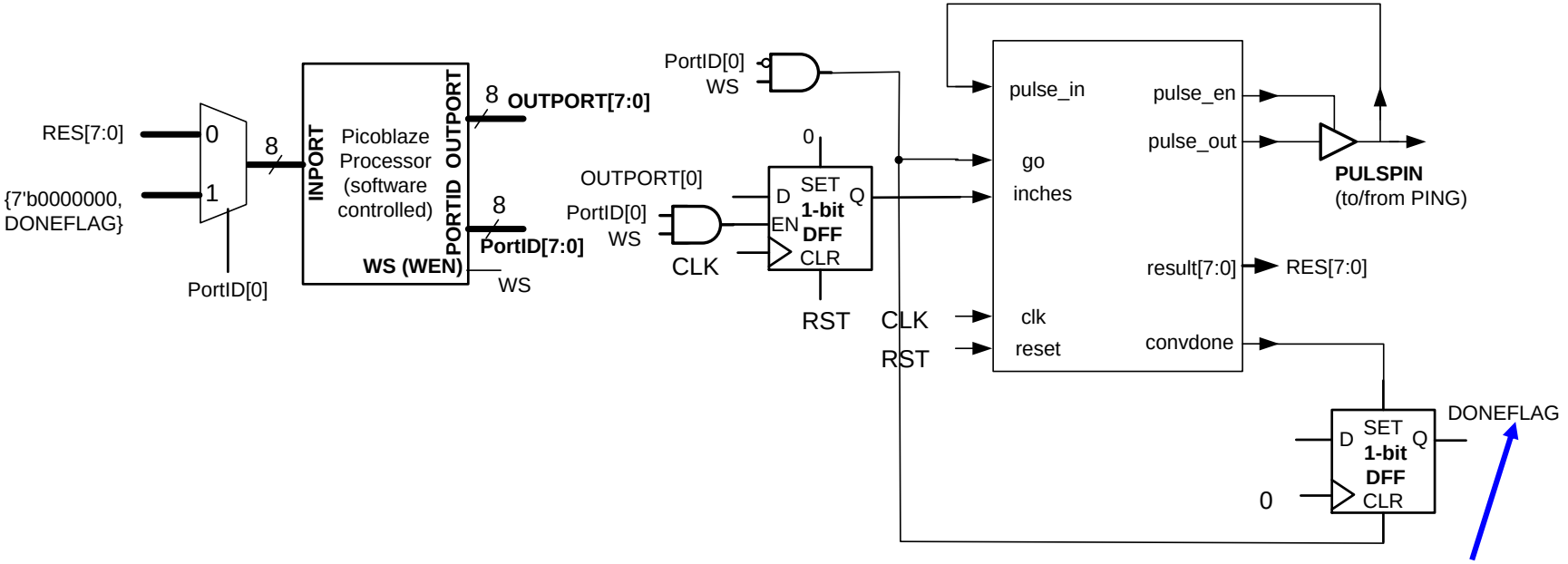
High-Order Interleaving

- General strategy is to place devices at ranges of address divide by **more-significant** bits

Dec	A7	A6	A5	A4	A3	A2	A1	A0	Assigned Device
00	0	0	0	0	0	0	0	0	64x8
01			0	0	0	0	0	1	Memory
…									
63			1	1	1	1	1	1	
…									open
128	1	0	0	0	0	0	0	0	8-bit Register
…									open
192		1	0	0	0	0	0	0	1-bit D-FF
…									open

PING))) Interfacing

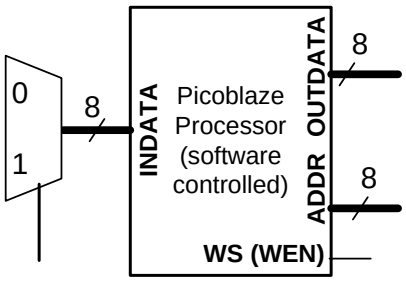
- Work with your instructor to explore alternatives for interfacing the PING))) engine you created to the Picoblaze processor
  - Output (Write): PortID (address) 0 => Go, PortID (address) 1 => Inches/cm
  - Input (Read): PortID (address) => Result, PortID (address) 1 => Done

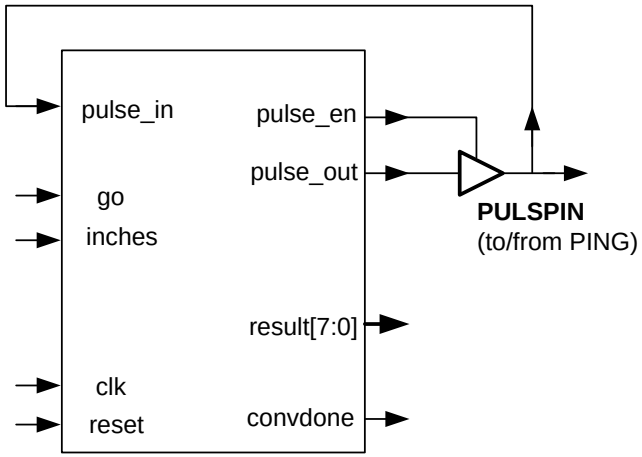


convdone only lasts 1 clock but we need to keep it at 1 until the software "sees" it. So register it. This could be a little state machine.

PING))) Interfacing

- Explore other alternatives…





PICOBLAZE ASSEMBLY

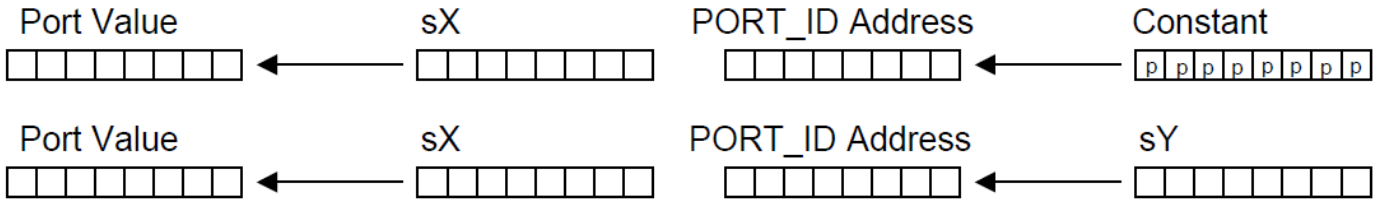
Relevant Manual Pages

- Pages 4-6, (7), 8, (9-11)
- Pages 16-36, focus on
  - OUTPUT 34
  - INPUT 35
  - SHIFTS 32,33
  - JUMP 17
  - LOAD 22
  - COMPARE 31
- Input/Output design 65-68

Output Instruction

OUTPUT

The OUTPUT instruction enables the contents of any register to be transferred to logic external to KCPSM3. The port address (in the range 00 to FF) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.



The user interface logic is required to decode the PORT_ID port address value and capture the data provided on the OUT_PORT. The WRITE_STROBE is set during an output operation (see 'READ and WRITE STROBES'), and should be used to clock enable the capture register or write enable a RAM (see 'Design of Output Ports').

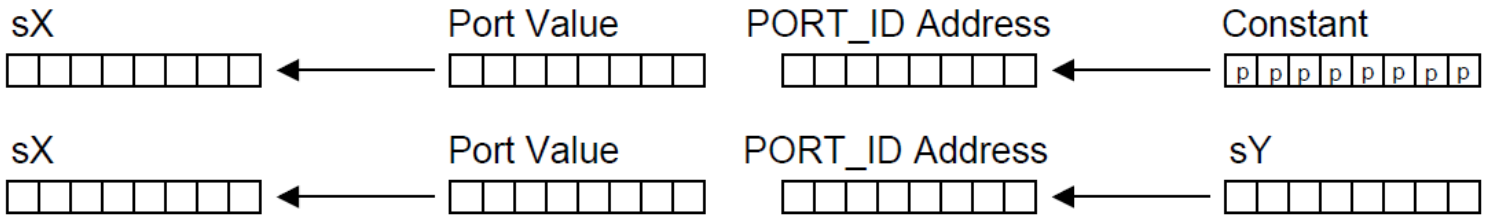
- Example: **`output s1, FF`**
  - Outputs the 8-bit number in **`s1`** as data on **`out_port`** to the address (**`port_id`**) of 0xFF

Taken from the KCPSM3 Manual

Input Instruction

INPUT

The INPUT instruction enables data values external to KCPSM3 to be transferred into any one of the internal registers. The port address (in the range 00 to FF) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.



The user interface logic is required to decode the PORT_ID port address value and supply the correct data to the IN_PORT. The READ_STROBE is set during an input operation (see 'READ and WRITE STROBES'), but it is not always necessary for the interface logic to decode this strobe. However, it can be useful for determining when data has been read, such as when reading a FIFO buffer (see 'Design of Input Ports').

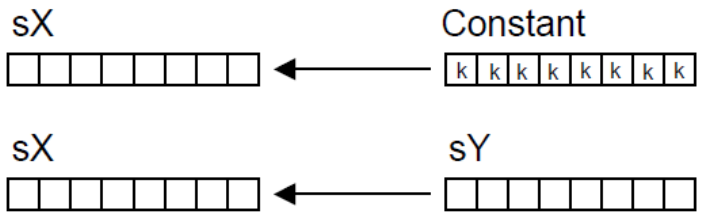
- Example: **`input  s8, 0c`**
  - Places the address 0x0c on the **`port_id`** and then grabs data from the **`in_port`** at the end of the second cycle and writes it into register **`s8`**

Taken from the KCPSM3 Manual

LOAD Instruction

LOAD

The LOAD instruction provides a method for specifying the contents of any register. The new value can be a constant, or the contents of any other register. The LOAD instruction has no effect on the status of the flags.



Since the LOAD instruction does not effect the flags it may be used to reorder and assign register contents at any stage of the program execution. The ability to assign a constant with no impact to the program size or performance means that the load instruction is the most obvious way to assign a value or clear a register.

The first operand of a LOAD instruction must specify the register to be loaded as register 's' followed by a hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

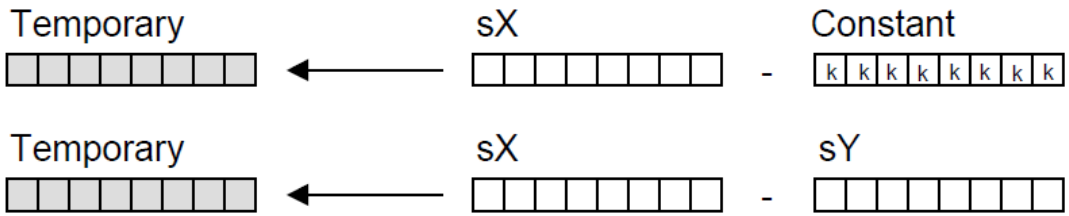
- Example: **`load  s3, a5`**
  - Loads the constant 0xa5 into register **`s3`**

Taken from the KCPSM3 Manual

Compare Instruction

COMPARE

The COMPARE instruction performs an 8-bit subtraction of two operands Unlike the 'SUB' instruction, the result of the operation is discarded and only the flags are affected. The ZERO flag is set when all the bits of the temporary result are low and indicates that both input operands were identical. The CARRY flag indicates when an underflow has occurred and indicates that the second operand was larger than the first. For example, if 's05' contains 27 hex and the instruction COMPARE s05,35 is performed, then the CARRY flag will be set (35>27) and the ZERO flag will be reset (35≠27).



CARRY ? Set if 'sY' or 'kk' is **greater than** 'sX'. Reset in all other cases.

ZERO ? Set if operands are **equal**. Reset in all other cases.

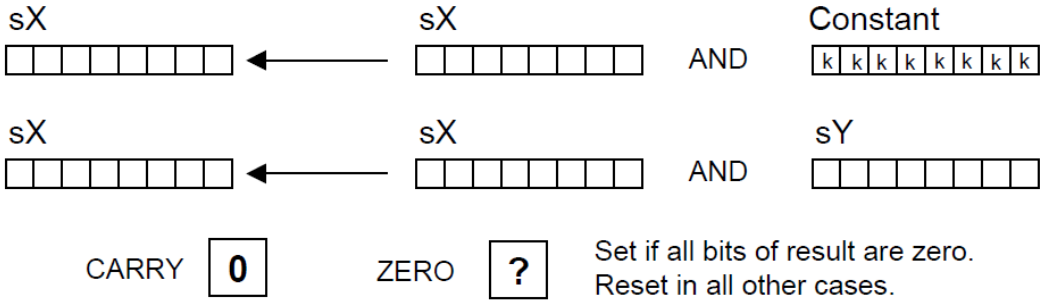
- Example: **`compare sf, 2a`**
  - Compares the data in register **`sf`** to the hex constant **`0x2a`**. It sets the Z flag (to determine equality) and C flag (to indicate less-than)

Taken from the KCPSM3 Manual

AND Instruction

AND

The AND instruction performs a bit-wise logical 'AND' operation between two operands. For example 00001111 AND 00110011 will produce the result 00000011. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The AND operation is useful for resetting bits of a register and performing tests on the contents (see also TEST instruction). The status of the ZERO flag will then control the flow of the program.



Each AND instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

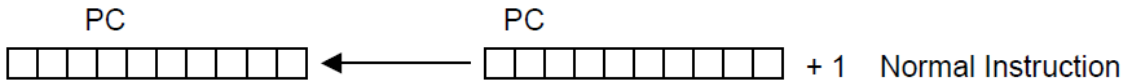
- Example: **`and s3, 04`**
  - Takes the bitwise AND of the data in register **`s3`** and the hex constant **`0x04`**, overwriting s3 with the result. It **sets the Z flag if the result of the ANDing is 0**

Taken from the KCPSM3 Manual

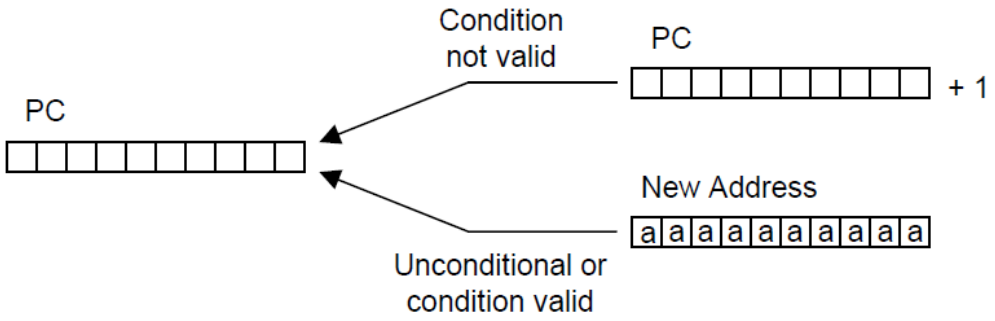
Jump Instruction

JUMP

Under normal conditions, the program counter (PC) increments to point to the next instruction. The address space is fixed to 1024 locations (000 to 3FF hex) and therefore the program counter is 10 bits wide. It is worth noting that the top of memory is 3FF hex and will increment to 000.



The JUMP instruction may be used to modify this sequence by specifying a new address. However, the JUMP instruction may be conditional. A conditional JUMP will only be performed if a test performed on either the ZERO flag or CARRY flag is valid. The JUMP instruction has no effect on the status of the flags.




```
JUMP aaa
JUMP Z,aaa
JUMP NZ,aaa
JUMP C,aaa
JUMP NC,aaa
```

- Example: **`jump Z, label`**
  - Jumps to the location specified by **`label`** if the condition bit (**Z**) is true

Taken from the KCPSM3 Manual