Programming The TramelBlaze using Assembly Code

CSULB
John Tramel

Instruction Set Architecture

- An instruction set architecture (ISA) is the abstract model of a computer
- The realization of an ISA is called an *implementation*
- The ISA defines everything a machine language programmer needs to know in order to program a computer
- Differences in the ISAs are the primary distinctions between different computer architectures

Microarchitecture

- The ISA presents the interface to the programmer but does not reveal how the computer has been implemented
- The microarchitecture is the set of processor design techniques that have been used to implement the ISA
  - Microprogrammed
  - Hard-wired control
- As an example the Intel Pentium and the Advanced Micro Devices Athlon processors implement nearly identical versions of the x86 instruction set, but have radically different internal designs

Emulators

- An emulator is hardware or software that enables one computer system to behave like another
- Early architecture differences between Apple computers and PC architectures cause software to run on one but not on the other. Emulators allowed the “look and feel” of the other - these were notoriously slow performers
- Many computer designs were built with the intention of running previous architectures (PDP-11) allowing reuse of existing software running on more powerful machines

Embedded Processor for SOC Design at CSULB

- The first SOC design classes taught at CSULB were targeting the Xilinx Spartan 3 family found on the Digilent Nexys 2 development boards
- The move to the Nexys 3 development boards brought with it the Xilinx Spartan 6 family
- Xilinx had developed an 8-bit embedded microcontroller, the PicoBlaze, that was natively hosted on the Nexys 3/6 families
- The move to the Nexys 4 with the Artix 7 family meant the PicoBlaze could not longer be utilized since there was not a version compatible with the Artix 7

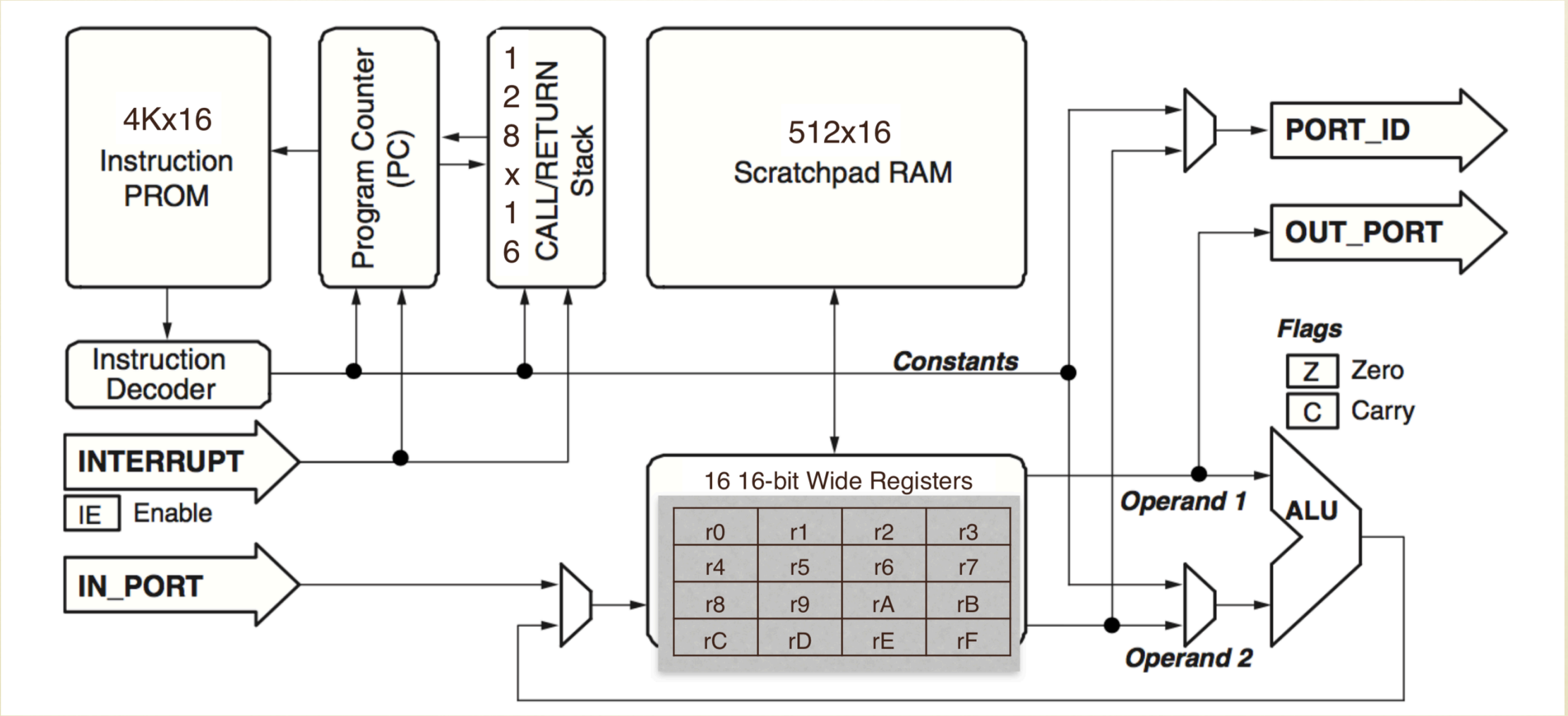
Embedded Processor for SOC Design at CSULB

- The solution that would enable the students to instantiate a relatively simple embedded microcontroller into their designs was the development of an *emulator* of the PicoBlaze, the TramelBlaze
- In order to allow all of Xilinx's documentation to remain valid the TramelBlaze implemented the PicoBlaze ISA
- The differences were that the TramelBlaze is a 16-bit architecture, 4K instructions (1K), 512 word Scratchpad RAM (64), 128 word Call/Return Stack (31)

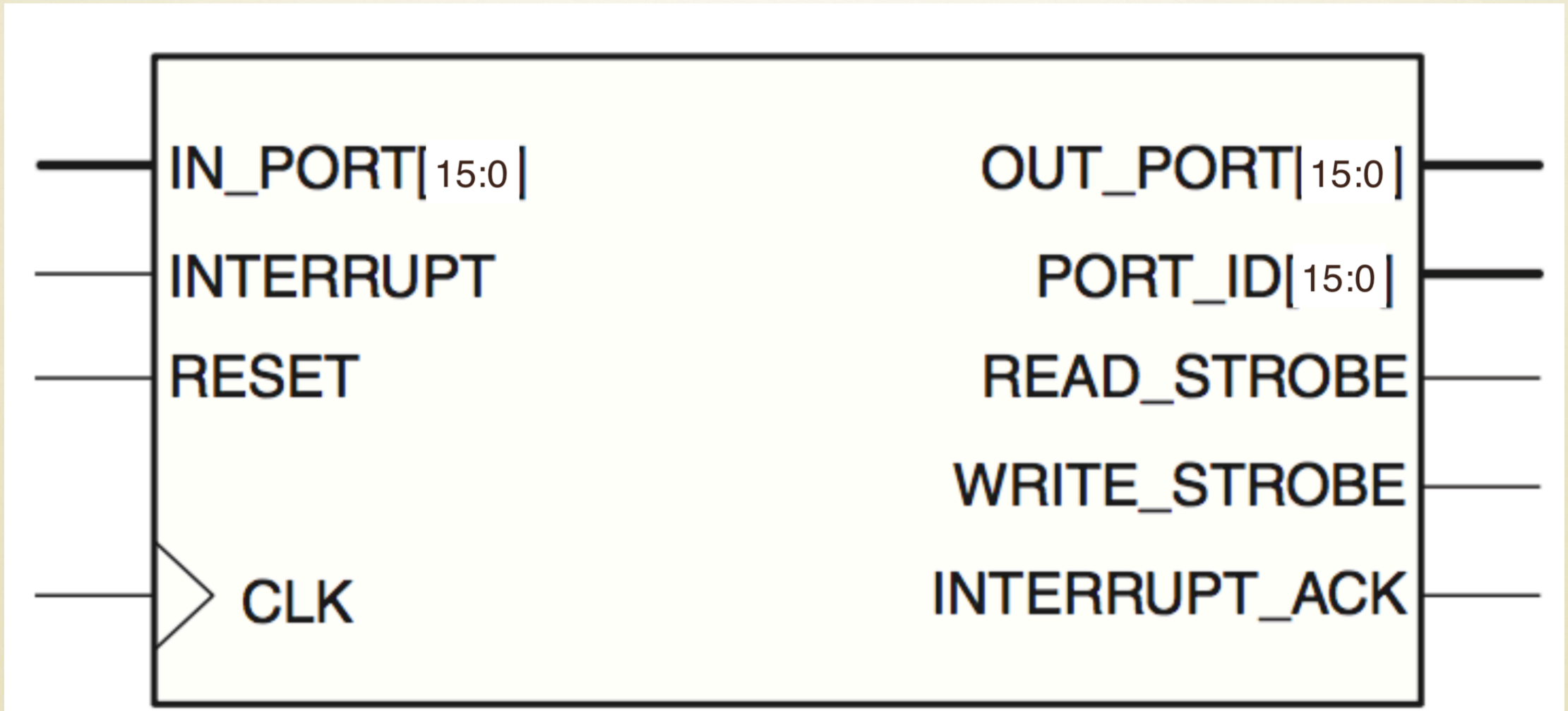
TramelBlaze

- The TramelBlaze is a 16-bit processor core designed to emulate the Xilinx PicoBlaze
- The ISA of the TramelBlaze is the ISA of the PicoBlaze
- It is written in Verilog and is instantiated in the users design as a module
- The assembler for the TramelBlaze ISA is written in Python and will generate the memory models required for building a Xilinx code ROM along with other files useful for debug

TramelBlaze Architecture



TramelBlaze Top-Level Block Diagram



Instantiating the TramelBlaze

```
tramelblaze_top tbt   (.CLK(clk),
                       .RESET(rst),
                       .IN_PORT(inport),
                       .INTERRUPT(interrupt),
                       .OUT_PORT(outport),
                       .PORT_ID(portid),
                       .READ_STROBE(readstrobe),
                       .WRITE_STROBE(writestrobe),
                       .INTERRUPT_ACK(interruptack)
                       );
```

- This documents the interface between the TramelBlaze and the rest of the SOPC design around it

TramelBlaze Instantiation within the TramelBlaze_top

```
tramelblaze tramelblaze
      (
      .CLK(CLK),
      .RESET(RESET),
      .IN_PORT(IN_PORT),
      .INTERRUPT(INTERRUPT),

      .OUT_PORT(OUT_PORT),
      .PORT_ID(PORT_ID),

      .READ_STROBE(READ_STROBE),
      .WRITE_STROBE(WRITE_STROBE),
      .INTERRUPT_ACK(INTERRUPT_ACK),

      .ADDRESS(ADDRESS),
      .INSTRUCTION(INSTRUCTION)
      );

tb_rom tb_rom (
      .clka(CLK),              // input clka
      .addra(ADDRESS),         // input  [11 : 0] addra
      .douta(INSTRUCTION)      // output [15 : 0] douta
      );
```

- The top has both the TramelBlaze architecture and the code ROM that holds all of the machine code that the processor will execute

TramelBlaze Microcontroller Functional Blocks I

- General-Purpose Registers - The micro controller includes 16 16-bit wide registers, r0 through rF. All register operations are completely interchangeable (no special function registers)
- 4096 Instruction Program Store - The code ROM provides storage for a good number of instructions. The instructions are either one or two words each. Once the program has been assembled the ROM may be built for simulation and synthesis

TramelBlaze Microcontroller Functional Blocks II

- Arithmetic Logic Unit (ALU) - the 16-bit wide ALU performs all microcontroller operations including:
  - Basic arithmetic operations such as addition/subtraction
  - Bitwise logic operations such as AND, OR, and XOR
  - Arithmetic compare and bitwise test operations
  - Comprehensive shift and rotate operations
  - Program control operation such as jump or call

TramelBlaze Microcontroller Functional Blocks III

- Flags - ALU operations may affect the ZERO and CARRY flags. The ZERO flag indicates the result of the previous operation was zero and the CARRY flag indicates various conditions (noted in documentation). The INTERRUPT_ENABLE flag enables interrupt operations
- 512-word Scratchpad RAM - a RAM within the processor with access provided by STORE and FETCH instructions. STORE copies the contents of a register to the RAM and FETCH copies the contents of the RAM to a register

TramelBlaze Microcontroller Functional Blocks IV

- INPUT/OUTPUT - the Input/Output ports extends the TramelBlaze's capabilities to allow interfacing (or connecting) to other FPGA logic. The addressing capability of the Input/Output instructions is 0 to 65535 (FFFF) which will require an external address decoder. INPUT operations move data from surrounding FPGA logic into a register and the OUTPUT operations move data from a register to the surrounding FPGA logic

TramelBlaze Microcontroller Functional Blocks V

- Program Counter - the PC points to the next instruction to be executed. Upon each instruction fetch the PC will be incremented to the next instruction's address. JUMP, CALL, RETURN, RETURNI instructions and interrupt or reset events will modify the contents of the PC.

- Program Flow Control - the normal execution sequence of the processor is one instruction and then the next. JUMP instructions transfer control to the specified address. CALL instructions transfer control to the specified address and will push the address of the next instruction on the stack so that a RETURN instruction will restore the sequence (subroutine call). When interrupts are enabled and an interrupt occurs control will transfer to FFE and will push the address of the next instruction onto the stack. RETURNI instruction will restore the sequence. (return from interrupt)

Why Use TramelBlaze?

	TramelBlaze Microcontroller	FPGA Logic
Strengths	* Easy to program, excellent for control and state machine applications * Resource requirements remain constant with increasing complexity * Re-uses logic resources, excellent for lower-performance functions	* Significantly higher performance * Excellent at parallel operations * Sequential vs. parallel implementation trade-offs optimize performance or cost * Fast response to multiple, simultaneous inputs
Weaknesses	* Executes sequentially * Performance degrades with increasing complexity * Program memory requirements increase with increasing complexity * Slower response to simultaneous inputs	* Control and state machine applications more difficult to program * Logic resources grow with increasing complexity

TramelBlaze Machine Language

- Computers are designed to do one thing: *fetch* and *execute* instructions
- These instructions are referred to as *machine* instructions
- Machine instructions are composed of 1's and 0's and are partitioned into fields that match the different elements within the computer
- Format of TramelBlaze machine instruction:

							Word 1																Word 2															
						HEX	2nd	OPCODE								register Y				register X				Immediate Data (kk/pp/aaa)															
	Instruction	Description	Function	ZERO	CARRY		15	14	13	12	11	10	9	8		7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
19	JUMPC aaa	Jump if CARRY to aaa	if CARRY PC <- aaa	-	-	26	1	0	1	0	0	1	1	0	38	0	0	0	0	0	0	0	0	aaa															

Bit to indicate
0: 1 word instruction
1: 2 word instruction

Opcode to be decoded

r0 .. rF

r0 .. rF

TramelBlaze Assembly Language

- The is a **one:many** relationship between a high-level language instruction and machine instructions
- There is a **one:one** relationship between an Assembly instruction and a machine instruction
- Assembly instructions are referenced by a mnemonic that is meant to convey the function of the instruction

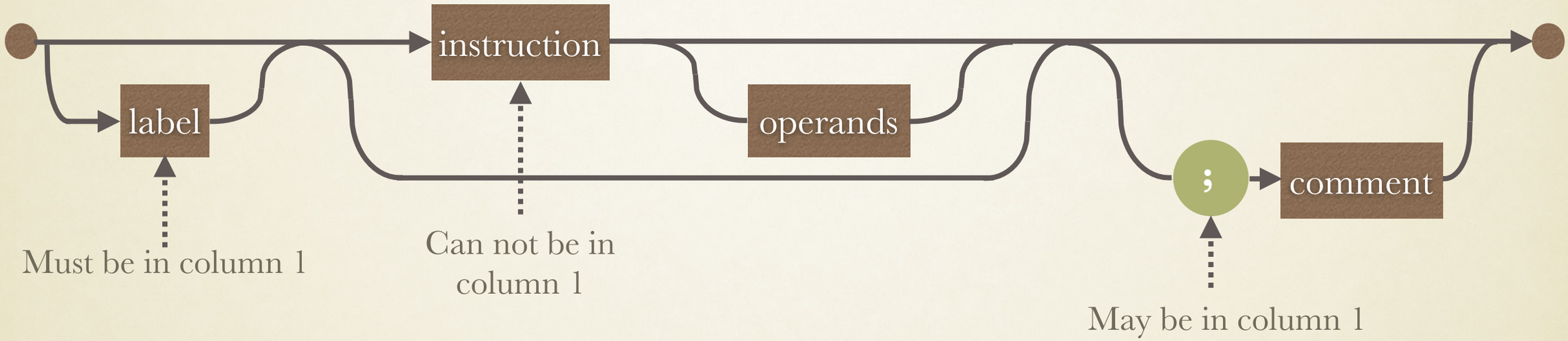
TramelBlaze Assembly Instructions

```
list =          ['NOP',       '0000','0000']
list = list +   ['ADD',       '8200','0400']
list = list +   ['ADDC',      '8600','0800']
list = list +   ['AND',       '8A00','0C00']
list = list +   ['CALL',      '8E00','0000']
list = list +   ['CALLC',     '9000','0000']
list = list +   ['CALLNC',    '9200','0000']
list = list +   ['CALLZ',     '9400','0000']
list = list +   ['CALLNZ',    '9600','0000']
list = list +   ['COMP',      '9800','1A00']
list = list +   ['DISINT',    '1C00','0000']
list = list +   ['ENINT',     '1E00','0000']
list = list +   ['INPUT',     'A200','2000']
list = list +   ['JUMP',      'A400','0000']
list = list +   ['JUMPC',     'A600','0000']
list = list +   ['JUMPNC',    'A800','0000']
list = list +   ['JUMPZ',     'AA00','0000']
list = list +   ['JUMPNZ',    'AC00','0000']
list = list +   ['LOAD',      'AE00','3000']
list = list +   ['OR',        'B200','3400']
list = list +   ['OUTPUT',    'B800','3600']
list = list +   ['RETURN',    '3A00','0000']
list = list +   ['RETURNC',   '3C00','0000']
list = list +   ['RETURNNC',  '3E00','0000']
list = list +   ['RETURNZ',   '4000','0000']
list = list +   ['RETURNNZ',  '4200','0000']
list = list +   ['RETDIS',    '4400','0000']
list = list +   ['RETEN',     '4600','0000']
list = list +   ['RL',        '4800','0000']
list = list +   ['RR',        '4A00','0000']
list = list +   ['SL0',       '4C00','0000']
list = list +   ['SL1',       '4E00','0000']
list = list +   ['SLA',       '5000','0000']
list = list +   ['SLX',       '5200','0000']
list = list +   ['SR0',       '5400','0000']
list = list +   ['SR1',       '5600','0000']
list = list +   ['SRA',       '5800','0000']
list = list +   ['SRX',       '5A00','0000']
list = list +   ['SUB',       'DC00','5E00']
list = list +   ['SUBC',      'E000','6200']
list = list +   ['TEST',      'E400','6600']
list = list +   ['XOR',       'E800','6A00']
list = list +   ['END',       '0000','0000']
list = list +   ['FETCH',     'F000','7200']
list = list +   ['STORE',     'F400','7600']
```

Note: The TramelBlaze assembly code is case insensitive. When the file is parsed all lower case is converted to upper. (This is copied directly from the trambler assembler).

TramelBlaze Assembly Instruction Formats

Basic Syntax

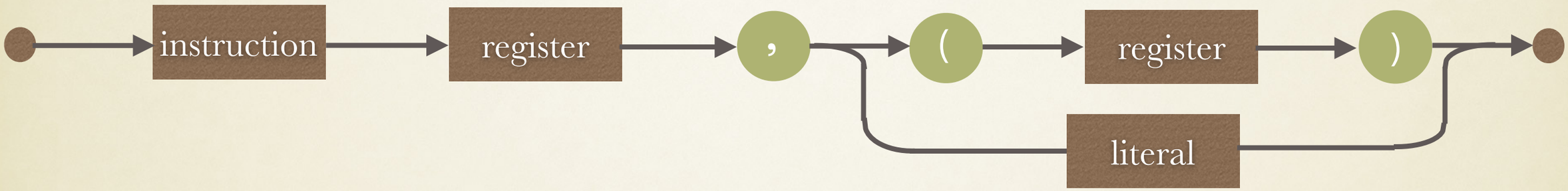


Register

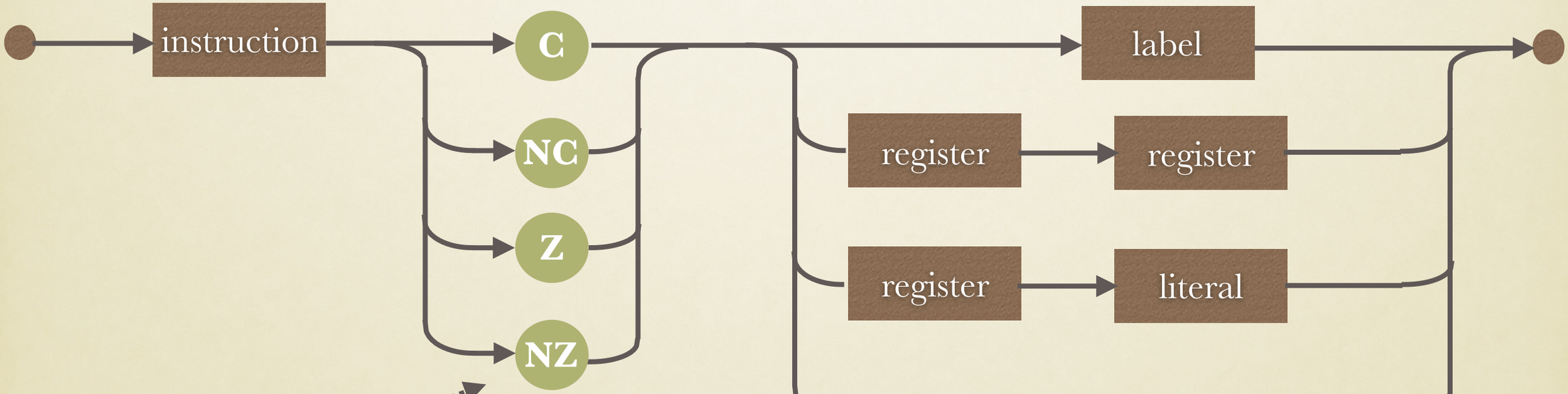
Immediate

TramelBlaze Assembly Instruction Formats

Indirect scratchpad / I/O port



Conditional



Append to the instruction
i.e. JUMPNZ

Assembler Directives

- Assembler directives are "instructions" added to the assembly language program that are intended to communicate with the assembler - they are not instructions to be converted to machine language
  - ADDRESS - default address is 0000. This directive changes where the instructions are placed in the range 000 … FFF
  - EQU - Equate a label with a constant value to be substituted prior to final assembly
  - END - End of the source code

```
ZEROS                EQU  0000
ONE                  EQU  0001
SIZE                 EQU  0200
MYREG                EQU  R9
```

```
; ISR vectored through 0FFE

                     ADDRESS 0FFE

ENDIT                JUMP ISR

                     END
```

- *EQU - allows user to define labels for reference in code*
- *ADDRESS - allows definition of assembled address space*
- *END - documents end of source code*

Default Base

- The default base for the TramelBlaze assembly language is HEXADECIMAL
- All references to immediate values will be interpreted as HEX

```
OUTPUT R0, 7654    ;here if an error
OUTPUT R1, 7654    ;here if an error
```

Interpreted as HEX values

Address Spaces

- *Registers* - A set of 16 16-bit registers referred to as r0 .. rF or R0 .. RF (remember case insensitive)
  - All registers are initialized to 0000 at reset
- *Instruction Memory* - There is a 4K instruction space with every instruction using either one or two words. This memory is not accessible from within a program.
- *Scratchpad Memory* - 512 words of RAM available for intermediate values. Accessed with fetch/store instructions
- *I/O Ports* - 65536 possible ports available for the external hardware. These ports are accessed with input/output instructions
- *Call Stack* - The hardware call stack is maintained to track return address for subroutines and interrupt handling. You can have a call depth up to 127 subroutines. Errors here are not detected - so be careful.

Flags

- The TramelBlaze has two internal status flags that represent the results of ALU operations. They are used to document the result of an operation and influence to execution of conditional instructions
- The Z flag is set when the result of an operation is zero and is cleared otherwise
- The C flag is set when an arithmetic carry of borrow (subtraction) is generated. C flag is also set by the test and testcy instructions

Interrupts

- The TramelBlaze has a single interrupt input. When interrupts are enabled and this pin transitions high (and remains till acknowledged), the normal flow is suspended and the processor jumps to a pre-defined interrupt address: FFE
- The assembly code must provide a JUMP instruction at FFE with FFF holding the address of the interrupt service routing
- Interrupts are controlled with the *enint* and *disint* instructions. *retdis* will return from an ISR with interrupts disabled and *reten* will return from an ISR with interrupts enabled

```
; john tramel
; cecs 460
; TramelBlaze Assembly Program Template

; declare constants for coding

ZEROS   EQU     0000
ONE     EQU     0001
CONST   EQU     O123
LABEL   EQU     ABCD

; the start of the assembly program should
; be used to initialize the design
; only executed at startup (reset)

START   LOAD    R1, LABEL
        LOAD    R2, 4567
        ADD     R1, R2
        SUBC    R1, R2
        JUMP    MAIN

; two architectural approaches
; 1) main follows init then routines
; 2) routines follow init main at bottom
; - prefer #2 since alway find main quickly

SUB1    LOAD    R5, 000F
        RETURN

SUB2    LOAD    R5, 000D
        RETURN

ISR     LOAD    R0, ONE
        INPUT   RA, 1234
        AND     RA, R0
        JUMPZ   ISR1
        RETEN

ISR1    SUB     R9, ONE
        RETEN

; main loop is where processor spends most of its time

MAIN    JUMP ENDIT

; ISR vectored through 0FFE

        ADDRESS 0FFE
ENDIT   JUMP ISR

        END
```

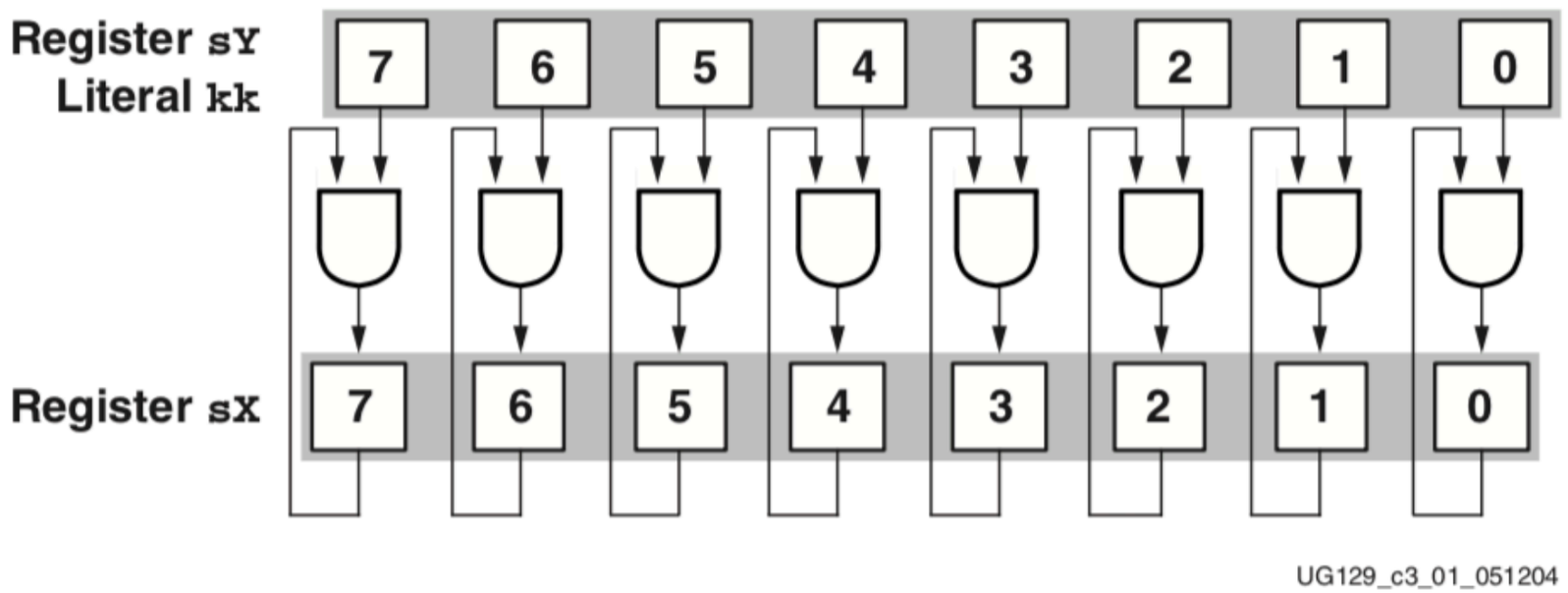
TramelBlaze Assembly Program Template

- This code is a template - don't try and make sense of the code
- Best to limit comments to header and before each major functional block - too many makes the code difficult to read
- Note the format of the code: 1) declarations, 2) startup code (initialization), 3) all service routines (plus ISR), 4) main loop
- Best to minimize time in ISR. Use flags and service in the routines - not in the ISR.

Processing Data
Bitwise AND, OR, XOR

- Destination register (sX) is always an operand
- Second operand is either another register (sY) or a literal (kk) that is included in the assembly program

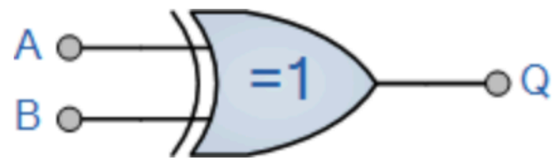
```
AND sX, sY
AND sX, kk
```




```
AND    R6, R7
AND    R9, FF00
```

Processing Data
Complement/Invert Register

- There is no NOT instruction - but knowing how an XOR instruction works allows us to perform a NOT

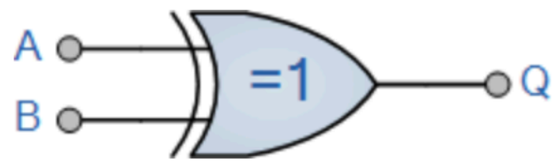
Symbol	Truth Table		
 2-input Ex-OR Gate	B	A	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	0
Boolean Expression Q = A ⊕ B	A **OR** B but NOT **BOTH** gives Q		

```
complement:
;   XOR sX, FF  invert all bits in register sX, same as one's complement

    LOAD s0, AA   ; load register    s0 = 10101010
    XOR  s0, FF   ; invert contents  s0 = 01010101
```

Processing Data
Complement/Invert Bit

- There is no NOT instruction - but knowing how an XOR instruction works allows us to complement a particular bit within a register

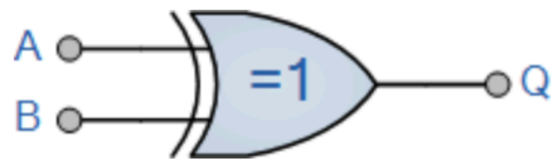
Symbol	Truth Table		
 2-input Ex-OR Gate	B	A	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	0
Boolean Expression Q = A ⊕ B	A **OR** B but NOT **BOTH** gives Q		

```
toggle_bit:
;   XOR sX, <bit_mask>

    XOR s0, 01  ; toggle the least-significant bit in register sX
```

Processing Data
Clearing A Register

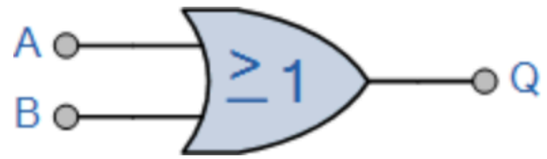
- There is no NOT instruction - but knowing how an XOR instruction works allows us to clear a register (fill with all zeros)

Symbol	Truth Table		
 2-input Ex-OR Gate	B	A	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	0
Boolean Expression Q = A ⊕ B	A **OR** B but NOT **BOTH** gives Q		

```
XOR sX, sX  ; clear register sX, set ZERO flag
```

Processing Data
Setting A Bit

- There is no instruction to set an individual bit in a register - but knowing how an OR instruction works allows us to set any bit within a register

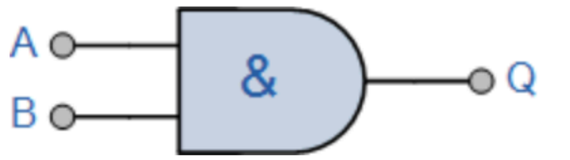
Symbol	Truth Table		
 2-input OR Gate	B	A	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	1
Boolean Expression **Q = A+B**	Read as A **OR** B gives Q		

```
set_bit:
;   OR sX, <bit_mask>

    OR s0, 01 ; set bit 0 of register s0
```

Processing Data
Clearing A Bit

- There is no instruction to clear an individual bit in a register - but knowing how an AND instruction works allows us to clear any bit within a register

Symbol	Truth Table		
 2-input AND Gate	B	A	Q
	0	0	0
	0	1	0
	1	0	0
	1	1	1
Boolean Expression **Q = A.B**	Read as A **AND** B gives Q		

```
clear_bit:
;   AND sX, <bit_mask>

    AND s0, FE  ; clear bit 0 of register s0
```

Processing Data Arithmetic Instructions

- ADD/ADDC - two add instructions computing the sum of two 16-bit operands (reg + reg) or (reg + literal)
  - Previous instructions may have set/cleared the CARRY bit and then utilized with the ADDC
- SUB/SUBC - two subtract instructions computing the difference of two 16-bit operands (reg-reg) or (reg-literal)
  - Previous instructions may have set/cleared the CARRY bit and then utilized with the SUBC

Processing Data Arithmetic Instructions

- Increment/Decrement - There are no instructions to increment or decrement but they may be implemented using either the ADD or SUB instructions

```
ONE     EQU    0001

        ADD    R0, ONE      // increment R0
        SUB    R0, ONE      // decrement R0
```

- Negate - There are no instructions to negate (change sign of a register). Knowing two's complement allows a simple methodology to implement a Negate.

```
NEGATE
        XOR    R0, FFFF     // Complement the register
        ADD    R0, ONE      // Increment results in -R0
```

Processing Data
Test and Compare Instructions

- Setting the CARRY Flag

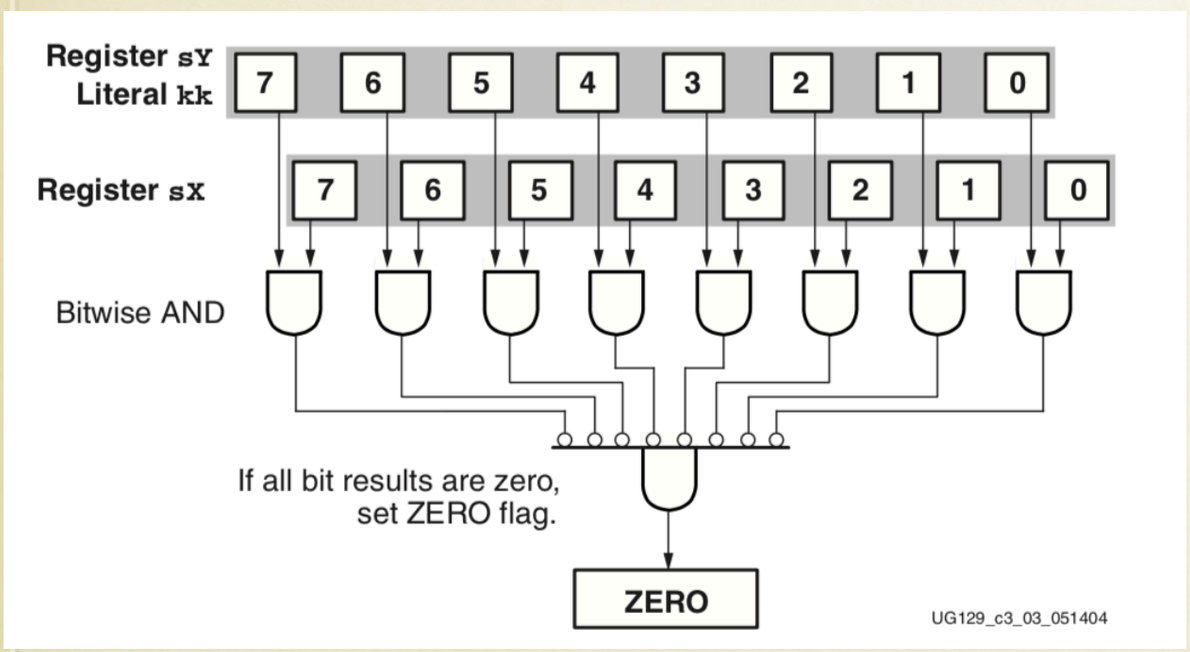
```
SET_CARRY
     LOAD    R0, 0000
     COMP    R0, 0001    // set CARRY clear ZERO
```

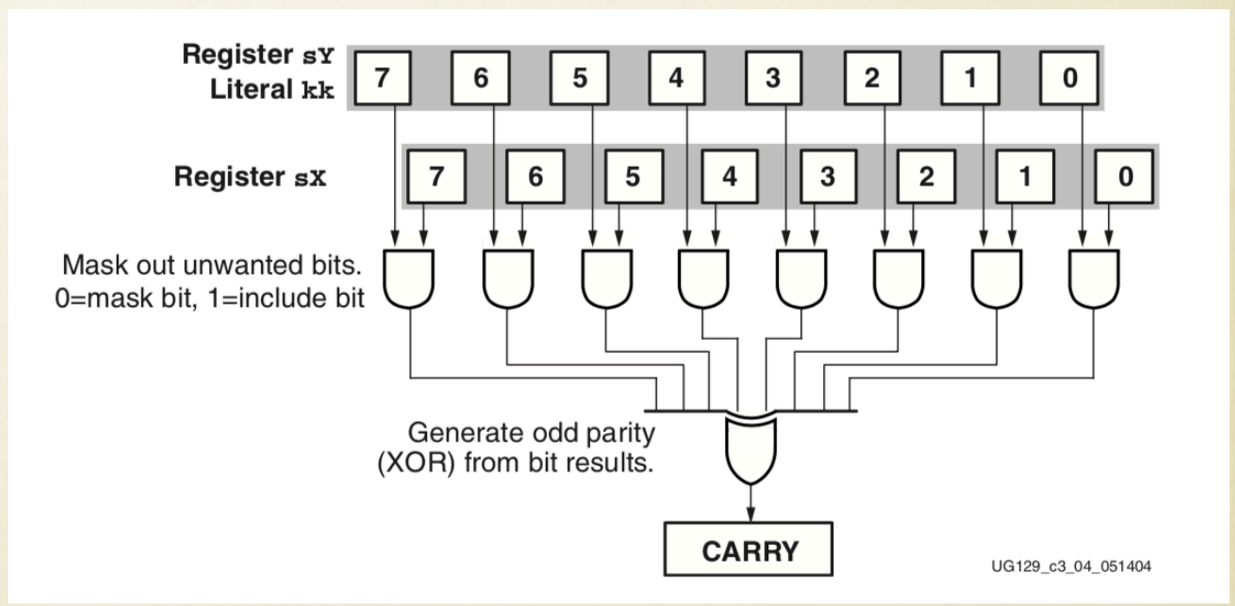
- Clearing the CARRY Flag

```
CLEAR_CARRY
     AND     R0, R0      // R0 unaffected, CARRY cleared
```

Processing Data
Test and Compare Instructions

- The TEST instruction performs bit testing via bitwise logical AND operation between two operands. Unlike the AND instruction, only the ZERO and CARRY flags are affected; no registers are modified. The ZERO flag is set if all the bitwise AND results are LOW.





Flag	When Flag=0	When Flag=1
ZERO	Operand_1 ≠ Operand_2	Operand_1 = Operand_2
CARRY	Operand_1 ≥ Operand_2	Operand_1 < Operand_2

Processing Data
Shift Instructions

	Shift Left ←		Shift Right →
SL0	Shift **L**eft with '**0**' fill. CARRY — Register sX — '0'	**SR0**	Shift **R**ight with '**0**' fill. '0' — Register sX — CARRY
SL1	Shift **L**eft with '**1**' fill. CARRY — Register sX — '1'	**SR1**	Shift **R**ight with '**1**' fill. '1' — Register sX — CARRY
SLX	**S**hift **L**eft, e**X**tend bit 0. CARRY — Register sX	**SRX**	**S**hift **R**ight, sign e**X**tend. Register sX — CARRY
SLA	**S**hift **L**eft through **A**ll bits, including CARRY. CARRY — Register sX	**SRA**	**S**hift **R**ight through **A**ll bits, including CARRY. Register sX — CARRY

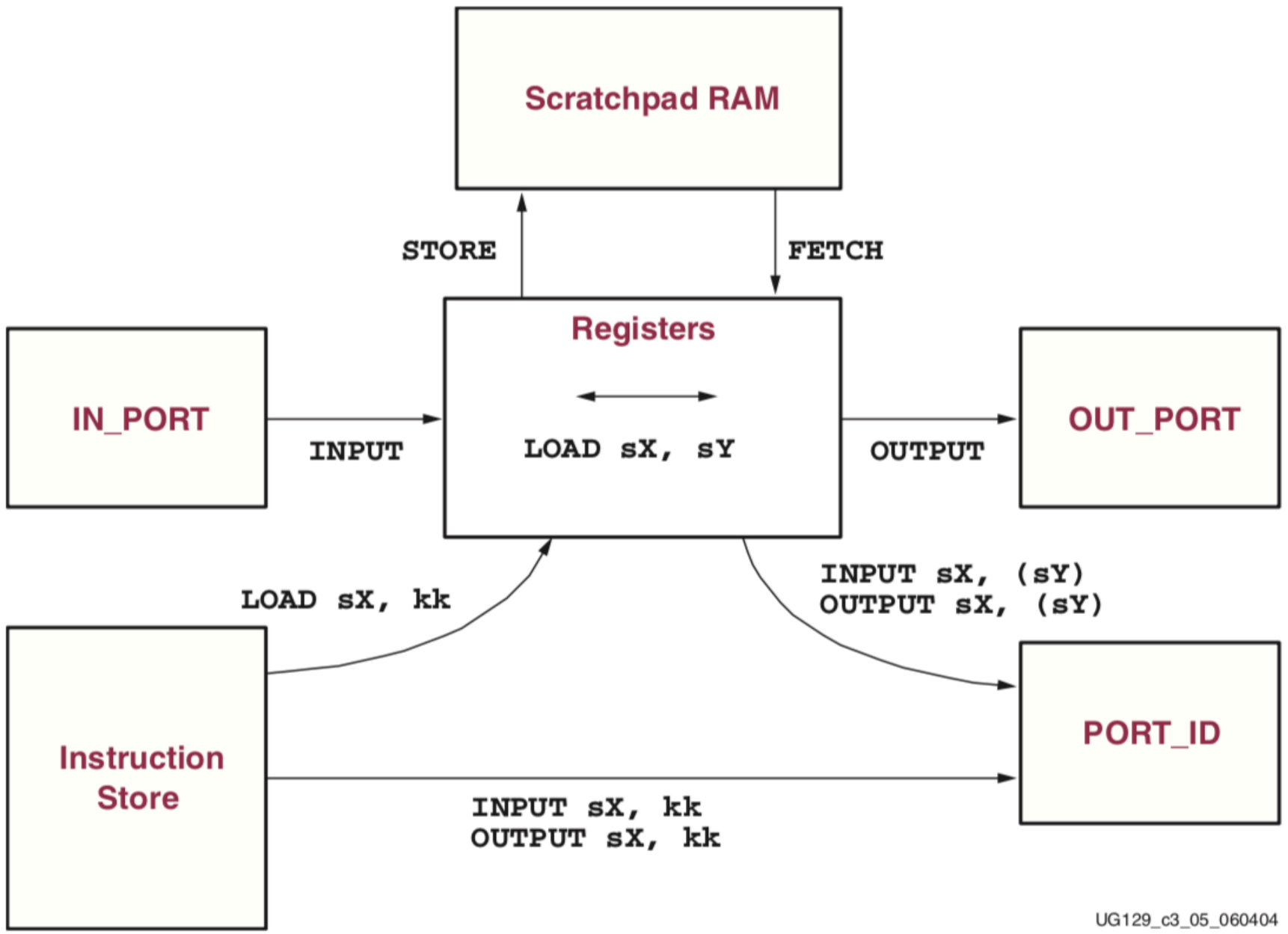
Remember: all registers in the TramelBlaze are 16-bits

Processing Data
Rotate Instructions

	Rotate Left		Rotate Right
RL		RR	

Remember: all registers in the TramelBlaze are 16-bits

Moving Data Within The TramelBlaze



Program Flow Control

- The Program Counter (PC) points to the next instruction to be executed and directly controls the TramelBlaze program flow
- By default the TramelBlaze proceeds to the next instruction in the instruction store (PC=PC+1/2)
- Three instructions may modify the program flow by loading the PC with a different value: JUMP, CALL & RETURN
- Each of these instructions may be conditionally executed

Condition	Description
<none>	Always true. Execute instruction unconditionally.
C	CARRY = 1. Execute instruction if CARRY flag is set.
NC	CARRY = 0. Execute instruction if CARRY flag is cleared.
Z	ZERO = 1. Execute instruction if ZERO flag is set.
NZ	ZERO = 0. Execute instruction if ZERO flag is cleared.

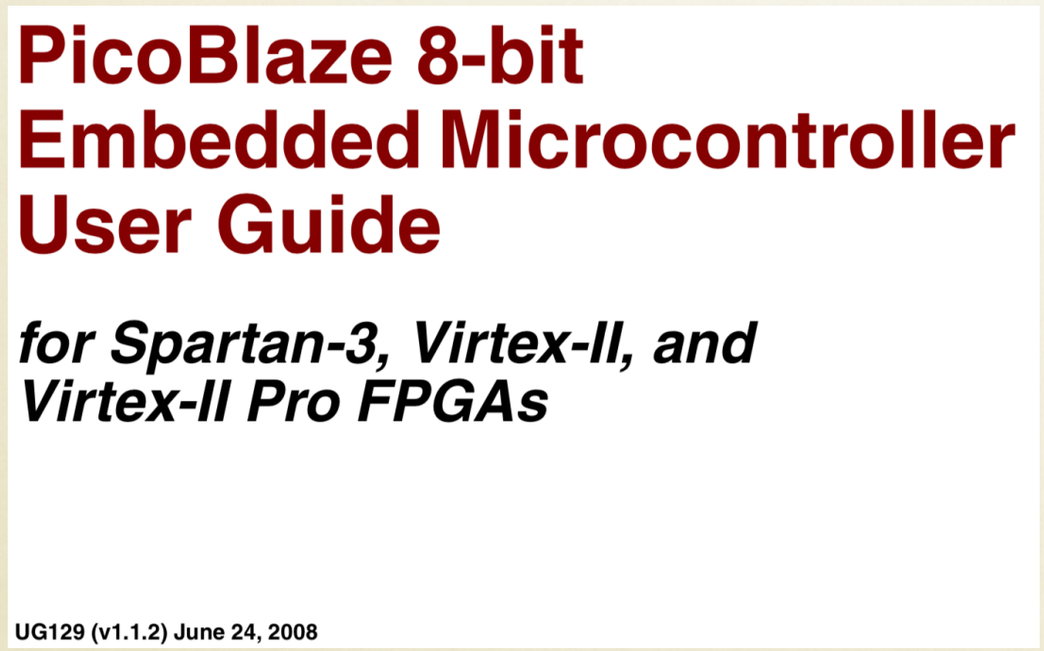
Program Flow Control

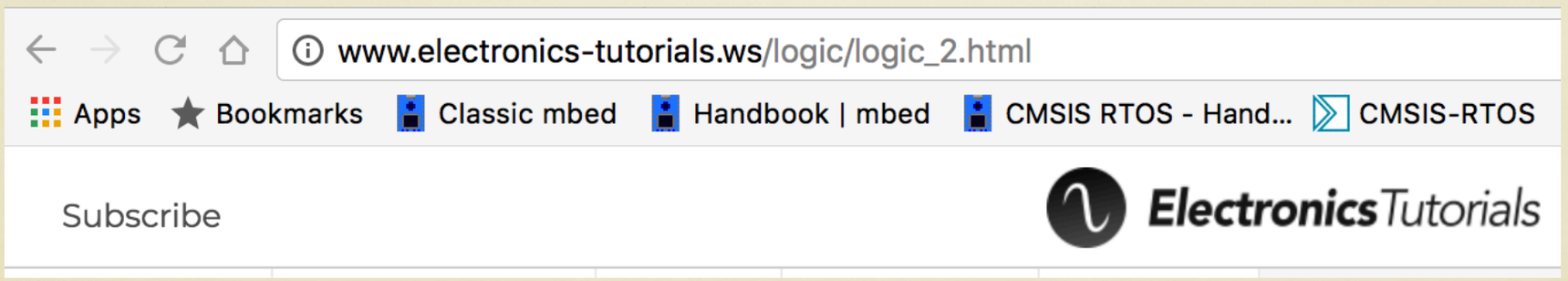
- The JUMP will have no effect if the condition is not met
- The JUMP instruction will not interact with the CALL/RETURN stack
- JUMP will transfer control to the destination address

- The CALL will have no effect if the condition is not met
- When executed the current PC (next address) will be pushed to the top of the CALL/RETURN stack and control will be transferred to the destination address
- When executed the RETURN instruction will pop the top of the CALL/RETURN stack to the PC and the next instruction executed will be the one following the CALL instruction

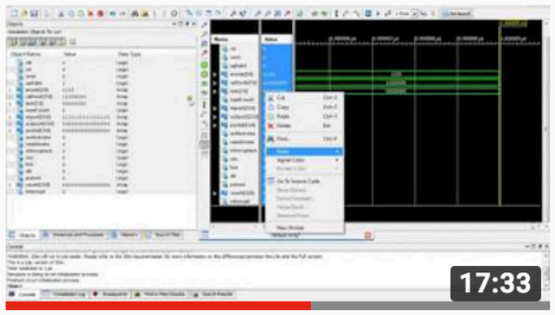
Credit

- Thanks to the following:





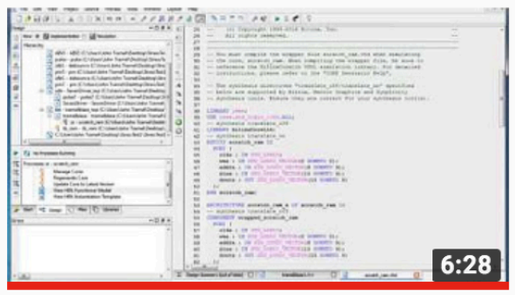
On-Line References (youtube.com)



TramelBlaze

John Tramel • 907 views • 1 year ago

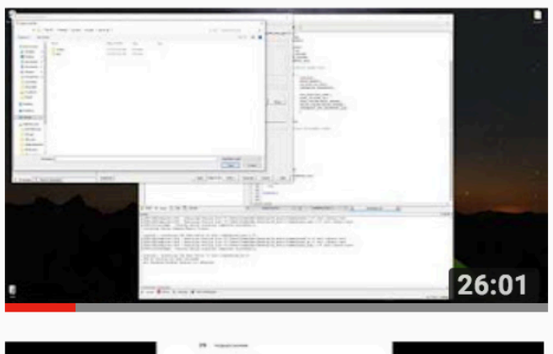
This video is about TramelBlaze.



ScratchRam

John Tramel • 271 views • 1 year ago

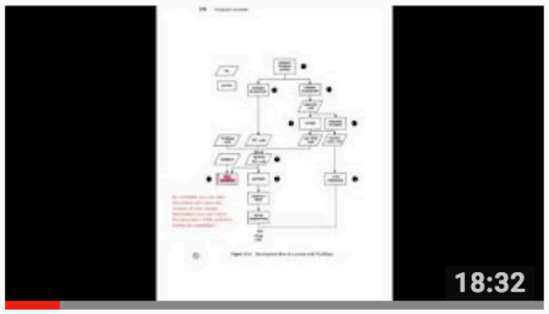
This video is about ScratchRam.



tramelblaze start synced

John Tramel • 560 views • 10 months ago

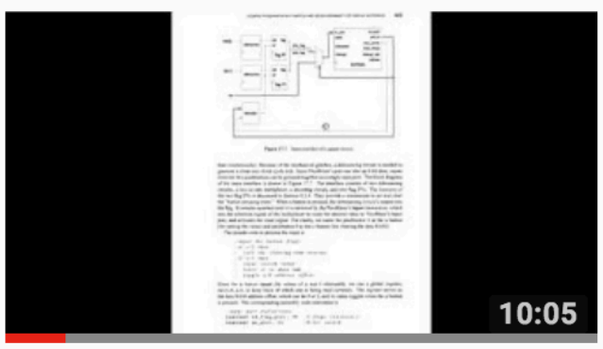
Description of creating memories for tramelblaze, simulating using tramelblaze, creating load file using the assembler - trambler.



PongChu_Ch15 1

John Tramel • 334 views • 1 year ago

This video is about PongChu_Ch15 1.



PongChu_CH17

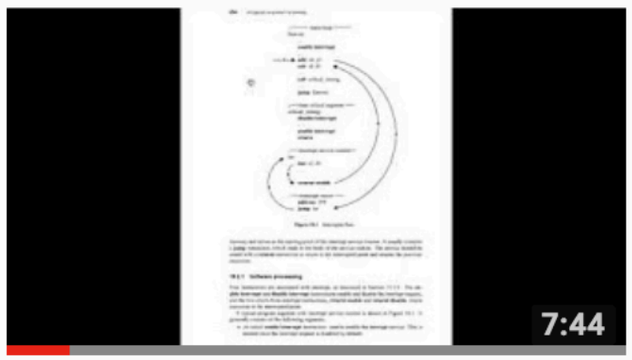
John Tramel • 203 views • 1 year ago

This video is about PongChu_CH17.

PongChu_CH16

John Tramel • 258 views • 1 year ago

This video is about PongChu_CH16.



PongChu_CH18

John Tramel • 166 views • 1 year ago

This video is about PongChu_CH18.