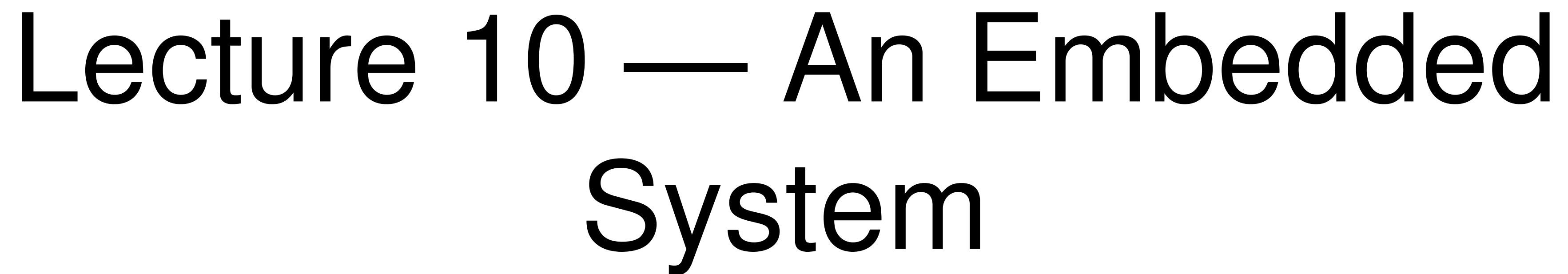
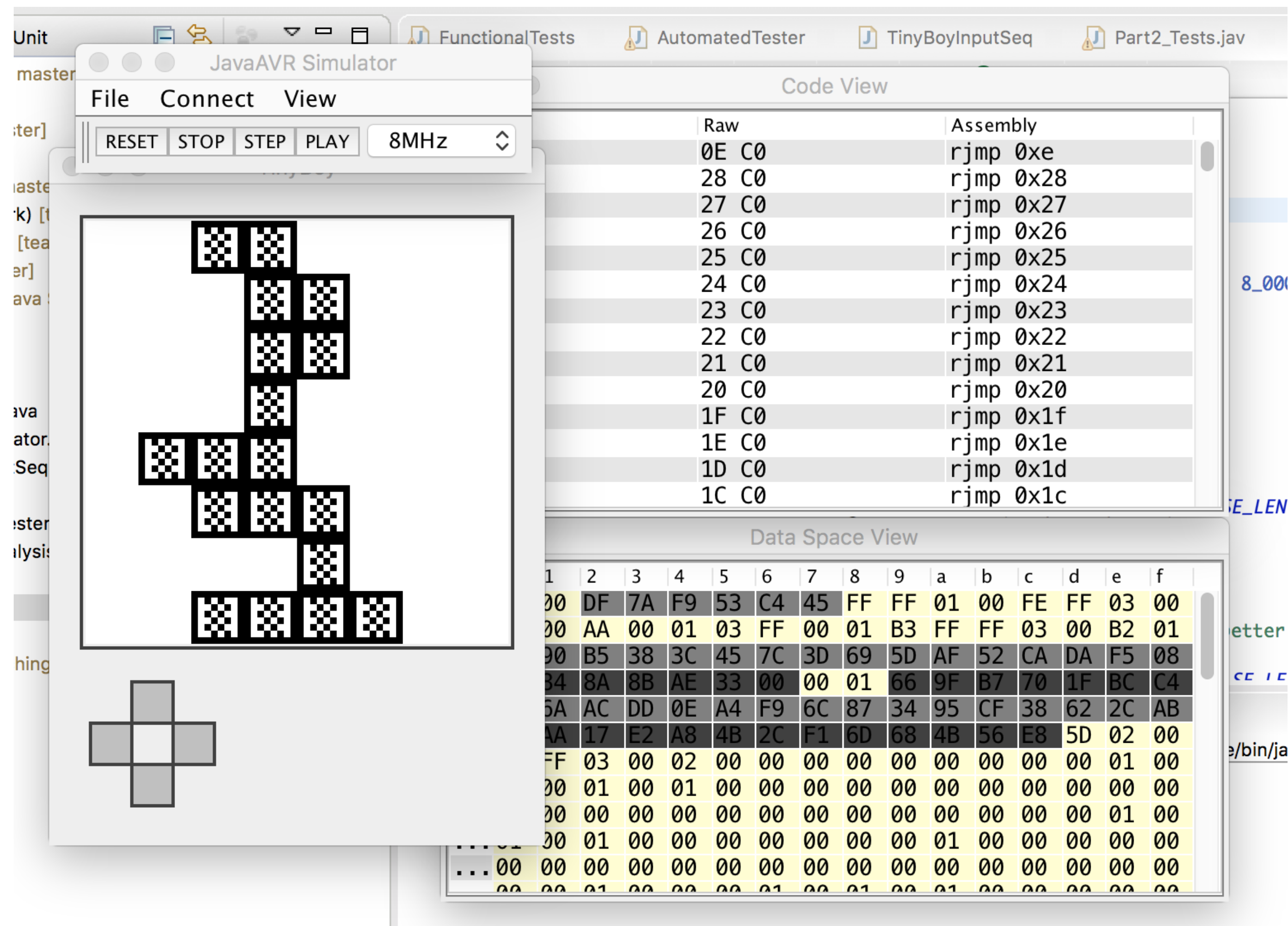


Lecture 10 — An Embedded System

David J. Pearce

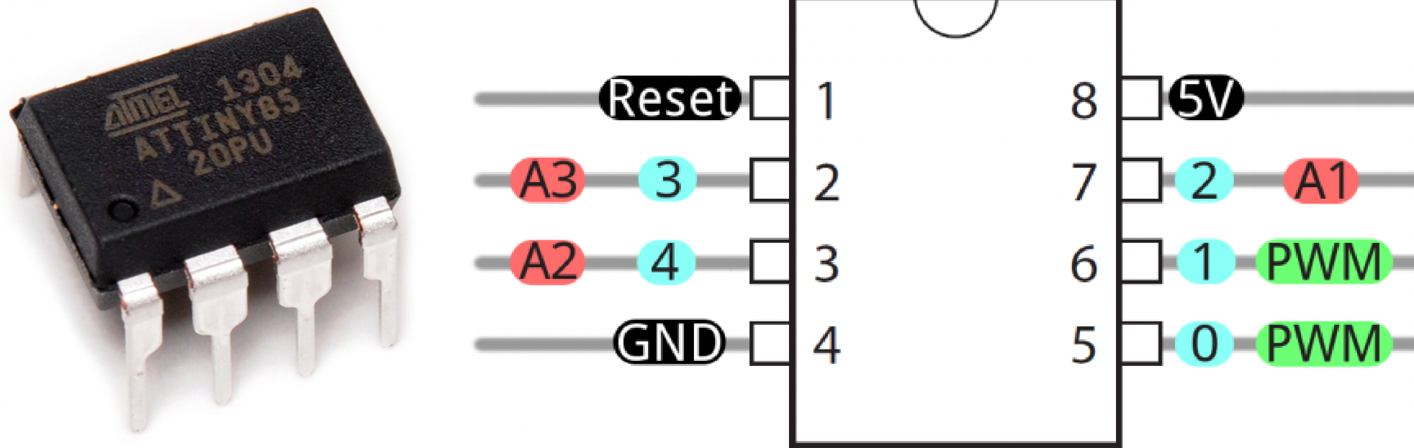
School of Engineering and Computer Science
Victoria University of Wellington

Assignment 2 — An Embedded System!



- Focused on **emulator** for ATtiny85

An Embedded System: ATtiny85



- An **8bit** microcontroller produced by Atmel
- Part of **AVR family** of microcontrollers (e.g. ATmega328)
- **Memory**: 8K flash; 512 bytes SRAM; 512 bytes EEPROM
- **Clock Rate**: max 20Mhz!

Machine Code vs Assembly Language

- Machine code is a binary format **directly executed** by microprocessor
- Generally speaking, humans don't read or write **machine code:**

```
0000 0000 0000 0000 6900 696e 2e74 0063
7263 7374 7574 6666 632e 5f00 4a5f 5243
...
```

- Normally, humans read and write **assembly language:**

```
    ...
    ldi r21,lo8(3)
    out 0x18, r1
    sbrs r24,0
    ...
```

- Assembly language is the **human readable** form of machine code

Hello World (JavaAVR console)

test.c

```
void spi_write(char c) {
  for(int i=0;i<8;++i) {
    PORTB = 0b00000000;
    if((c & 1) == 1) { PORTB = 0b00000011; }
    else { PORTB = 0b00000001; }
    c = c >> 1;
} }

int main (void){
  DDRB = 0b00000011; // set SCLK + MOSI to output
  char chars[] = "hello␣world";
  //
  for(int j=0;j!=11;++j) { spi_write(chars[j]); }
  //
  return 0;
}
```

Some AVR Instructions

`add r1, r2`	Add `r1` register to `r2` register (without carry) and assign `r1`
`andi r1, K`	Logical and between register `r1` and constant `K` and assign `r1`
`cp r1, r2`	Compare register `r1` against register `r2`
`ld r26, X`	Load one byte indirect from location in data space pointed to by register `X` into register `r26`.
`ldi r16, K`	Load 8-bit constant `K` to `r16` register
`mov r1, r2`	Copy register `r2` to `r1` register
`out A, r26`	Stores data from register r26 in the Register File to I/O Space
`rjmp 0x4`	Relative jump to an address within `PC + K + 1` words.
`sub r1, r2`	Substract register `r2` from register `r1` and assign `r1`

See *Atmel AVR Instruction Set Manual*

Data Space Layout

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0x0000	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	r13	r14	r15
0x0010	r16	r17	r18	r19	r20	r21	r22	r23	r24	r25	r26	r27	r28	r29	r30	r31
											X		Y		Z	

32 General Purpose Registers

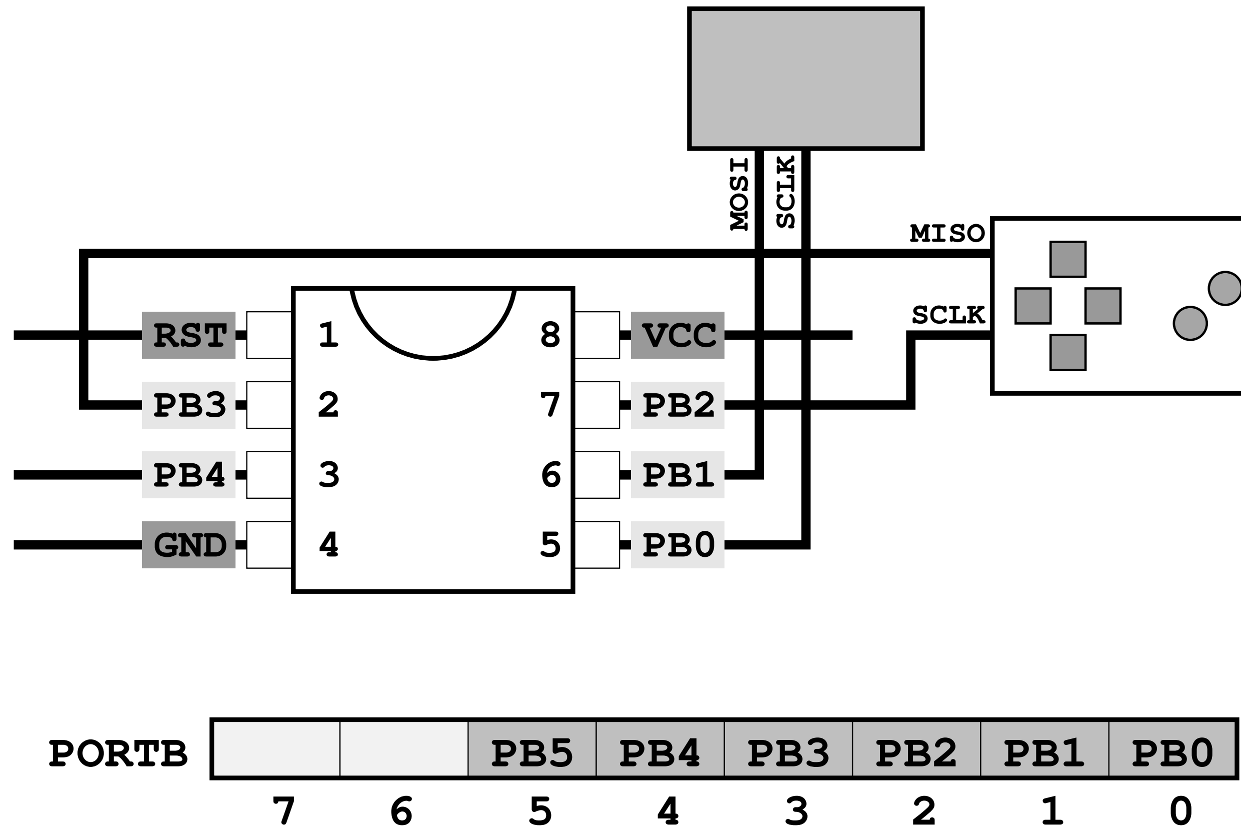
	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0x0020																
0x0030	PORTB														SP	
0x0040																
0x0050																

64 I/O Registers

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0x0060																
0x0070																
0x0080																
0x0090																
0x00A0																
0x00B0																
0x00C0																
0x00D0																
0x00E0																
0x00F0																

512 bytes SRAM

Input / Output



- Above shows **example** hookup of two SPI devices
- **Also:** DDRB (Data Direction) and PINB (Pins Address) registers

Status Register (SREG)

- **Bit 7 — I:** *Global Interrupt Enable*
- **Bit 6 — T:** *Bit Copy Storage*. Use for copying to / from individual register bits
- **Bit 5 — H:** *Half Carry Flag*
- **Bit 4 — S:** *Sign Flag*
- **Bit 3 — V:** *Overflow Flag*. For signed arithmetic, indicates whether an overflow occurred.
- **Bit 2 — N:** *Negative Flag*. For signed arithmetic, indicates whether result is negative.
- **Bit 1 — Z:** *Zero Flag*. For signed/unsigned arithmetic, indicates whether result is negative.
- **Bit 0 — C:** *Carry Flag*. Indicates a carry in an arithmetic or logic operation.

Conditional Branching

- Conditional branch (equality) implemented as follows:

```
cp r1, r2                /* compare r1 against r2 */
breq target              /* branch if equal (i.e. zero flag set) */
```

- Conditional branch (greater than or equal) implemented as follows:

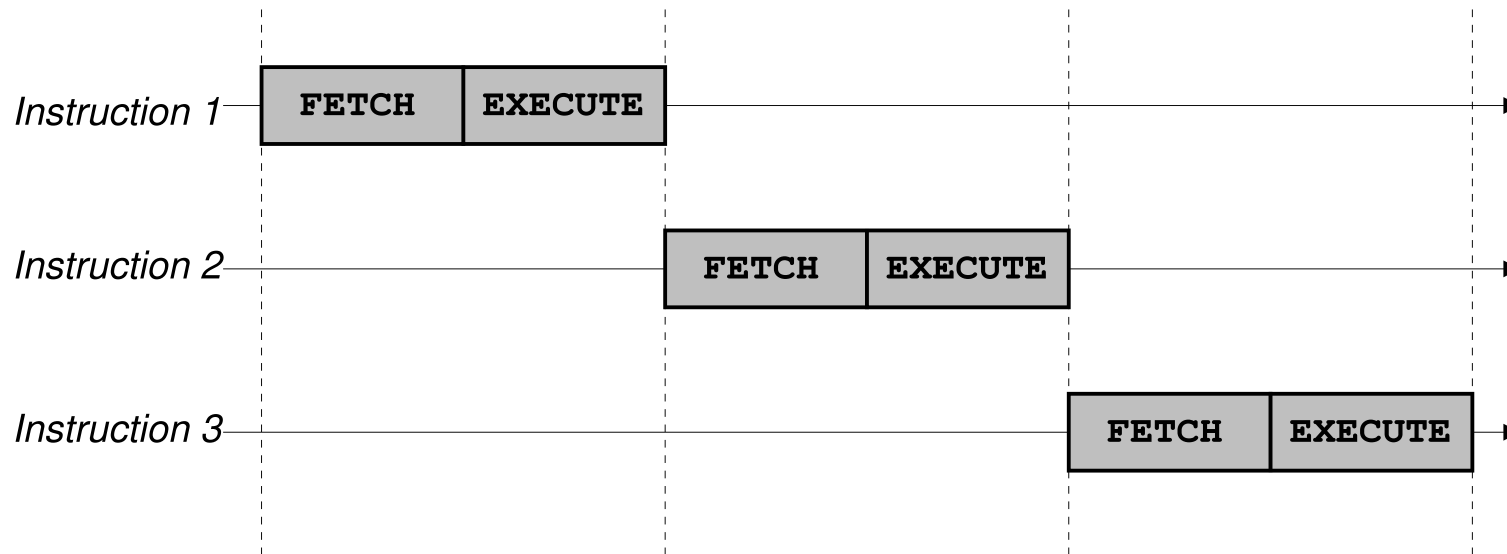
```
cp r26, r27             /* compare r26 against r27 */
brge target             /* branch if sign flag set */
```

- Conditional branch (not equals) implemented as follows:

```
cpi r2,123              /* compare r2 against 123 */
brne target             /* branch if zero flag not set */
```

- Notes:
  - » **Zero Flag** set after comparison if items equal
  - » **Sign Flag** set after comparison if left operand less than right

Instruction Timing



- Majority of instructions complete in **1 cycle**
- A few take **2 cycles** (e.g. `IJMP`, `LDS`, `RJMP`, `POP`, etc)
- Even fewer take **3 cycles or more** (e.g. `JMP`, `RCALL`, etc)

Code Generation

- Translation uses **registers** for intermediate results:

C	AVR

`int z = (7 + x) - y;`
`...`
	`adiw r24,7`
	`sub r24,r22`
	`sbc r25,r23`
	`...`

- Here, we have saved 4 instructions!
- Care required to avoid **clobbering values** stored in registers

Understanding the Stack

`push r12`	Push `r12` register onto stack
`pop r12`	pop byte off stack and assign to register `r12`

- Stack provided for additional **temporary storage**:

```
ldi r1, 255               /* store 255 in r1 */
ldi r2, 15                /* store 15 in r2 */
push r1                   /* push contents of r1 on stack */
push r2                   /* push contents of r2 on stack */
...
pop r4                    /* pop contents of stack into r4 */
```

- Stack grows **downwards**!
- Stack used primarily for **local variables**, and **return address**

Visualising the Stack

- Consider **executing** these instructions:

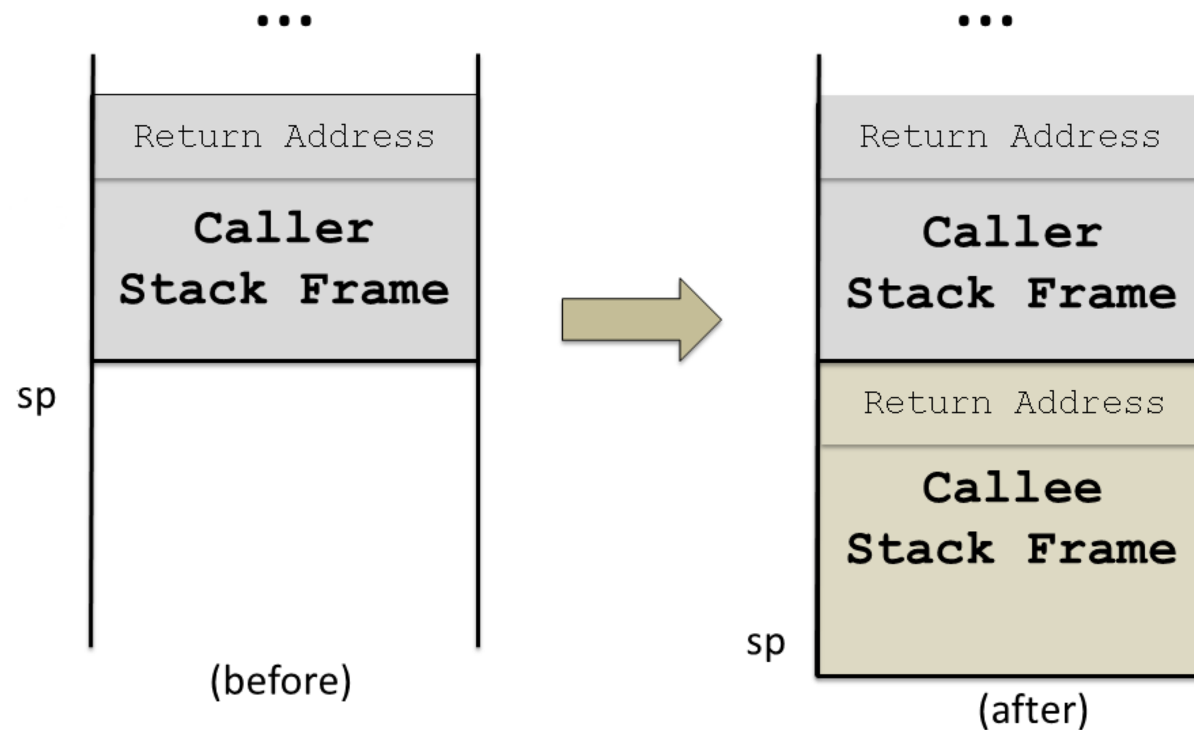
```
ldi r1, 255          /* store 255 in r1 */
ldi r2, 15           /* store 15 in r2 */
ldi r3, 01           /* store 1 in r3 */
push r1              /* push contents of r1 on stack */
push r2              /* push contents of r2 on stack */
push r3              /* push contents of r3 on stack */
```

- The effect on the stack can be **visualised** like so:

0x0220
0x0230
0x0240
0x0250
01
0F
FF

Call Stack

`call proc`	call procedure `proc` (`push pc + 2 ; jmp proc`)
`ret`	return from procedure (`pop pc`)



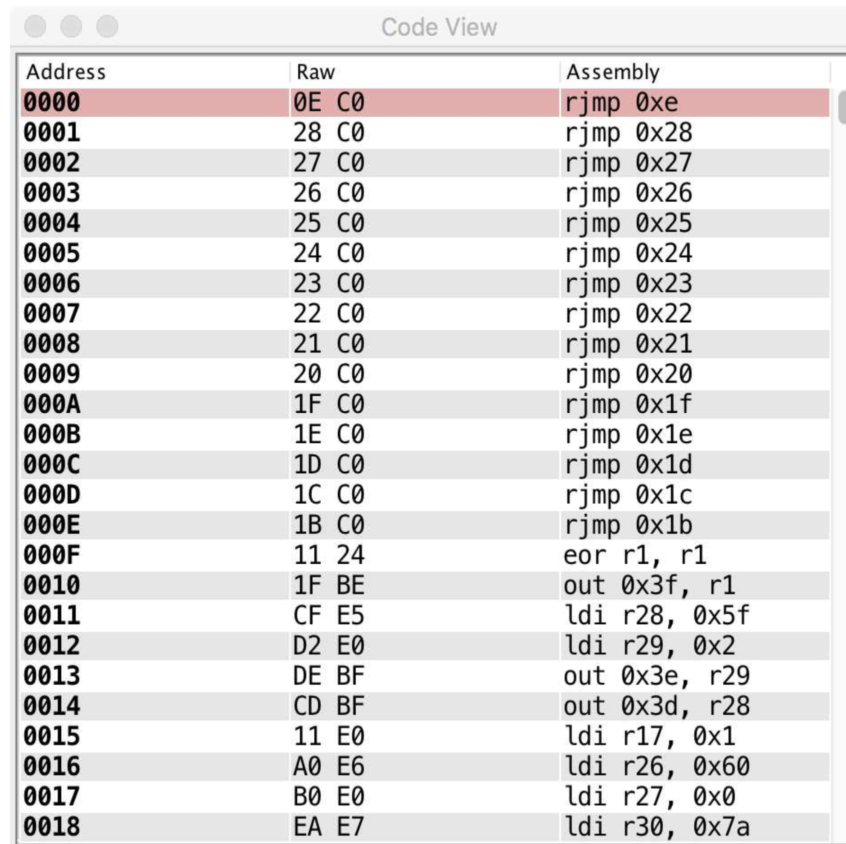
- Method calls implemented via **call** instruction
- This pushes address of **next instruction** then jumps to target

Interrupts

> “*In system programming, an* **interrupt** *is a signal to the processor emitted by hardware or software indicating an event that needs* **immediate attention***. An interrupt alerts the processor to a high-priority condition requiring the* **interruption of the current code** *the processor is executing.* ”
>
> –Wikipedia

- ATtiny has 15 **interrupt vectors**
- When interrupt triggered, control transfers to **vector location**

Interrupts



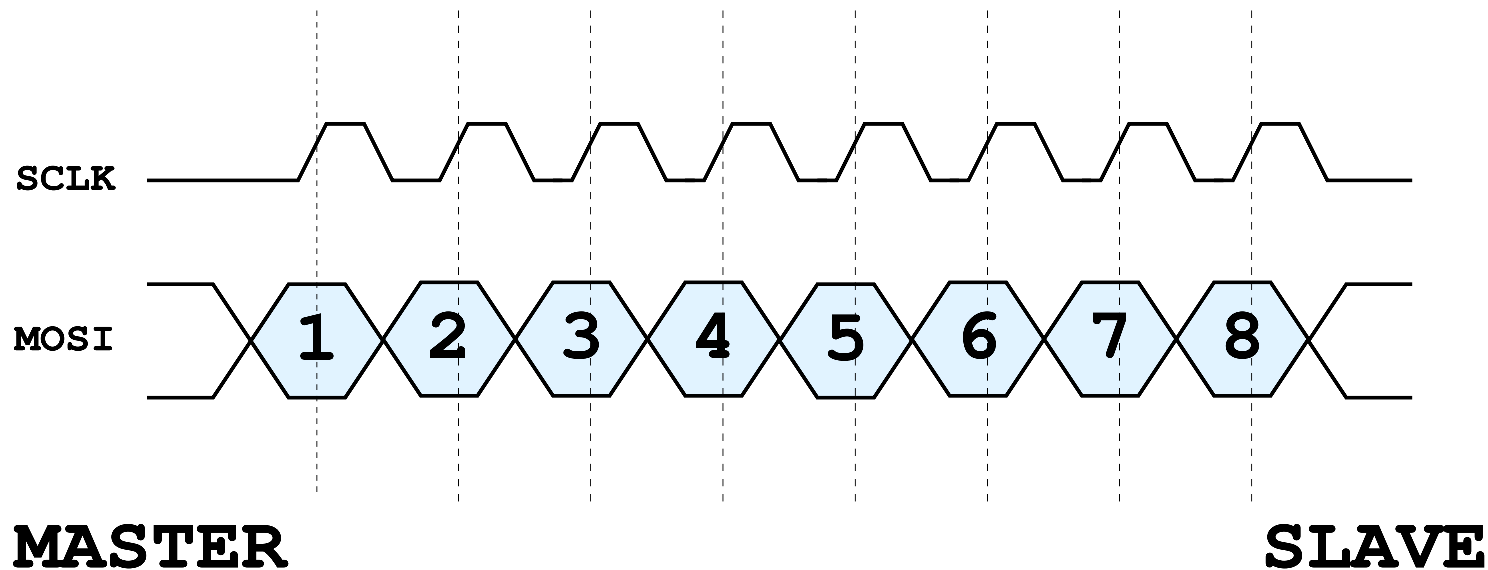
Code View

Address	Raw	Assembly
0000	0E C0	rjmp 0xe
0001	28 C0	rjmp 0x28
0002	27 C0	rjmp 0x27
0003	26 C0	rjmp 0x26
0004	25 C0	rjmp 0x25
0005	24 C0	rjmp 0x24
0006	23 C0	rjmp 0x23
0007	22 C0	rjmp 0x22
0008	21 C0	rjmp 0x21
0009	20 C0	rjmp 0x20
000A	1F C0	rjmp 0x1f
000B	1E C0	rjmp 0x1e
000C	1D C0	rjmp 0x1d
000D	1C C0	rjmp 0x1c
000E	1B C0	rjmp 0x1b
000F	11 24	eor r1, r1
0010	1F BE	out 0x3f, r1
0011	CF E5	ldi r28, 0x5f
0012	D2 E0	ldi r29, 0x2
0013	DE BF	out 0x3e, r29
0014	CD BF	out 0x3d, r28
0015	11 E0	ldi r17, 0x1
0016	A0 E6	ldi r26, 0x60
0017	B0 E0	ldi r27, 0x0
0018	EA E7	ldi r30, 0x7a

- Interrupt vectors occupy **bottom** of code space

Serial Peripheral Interface (SPI)

"The Serial Peripheral Interface bus (SPI) is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems." –Wikipedia



- **SCLK** (Serial Clock). On **rising edge**, slave reads MOSI
- **MOSI** (Master-Out, Slave-In). Carries bits from **master** to **slave**