ATMEGA128 Architecture and Assembly Programming Intro

Instructors:

Dr. Zhao Zhang (Sections A, B, C, D, E)

Dr. Phillip Jones (Sections F, G, J)

Overview of This Week

- Announcements
- Project Discussion
- Introduction to ATMEGA 128 Architecture and Assembly programming.

AVR ARCHITECTURE OVERVIEW

Why use assembly programming?

- Full access to hardware features
  - Compiler limits a programmers access to the hardware features that the compiler writer decided to implement
- Writing time critical portions of code
  - Allows tight control over what the CPU is doing on every clock cycle
- Debugging
  - It in not uncommon when trying to debug odd system behavior to have to look at disassembled code

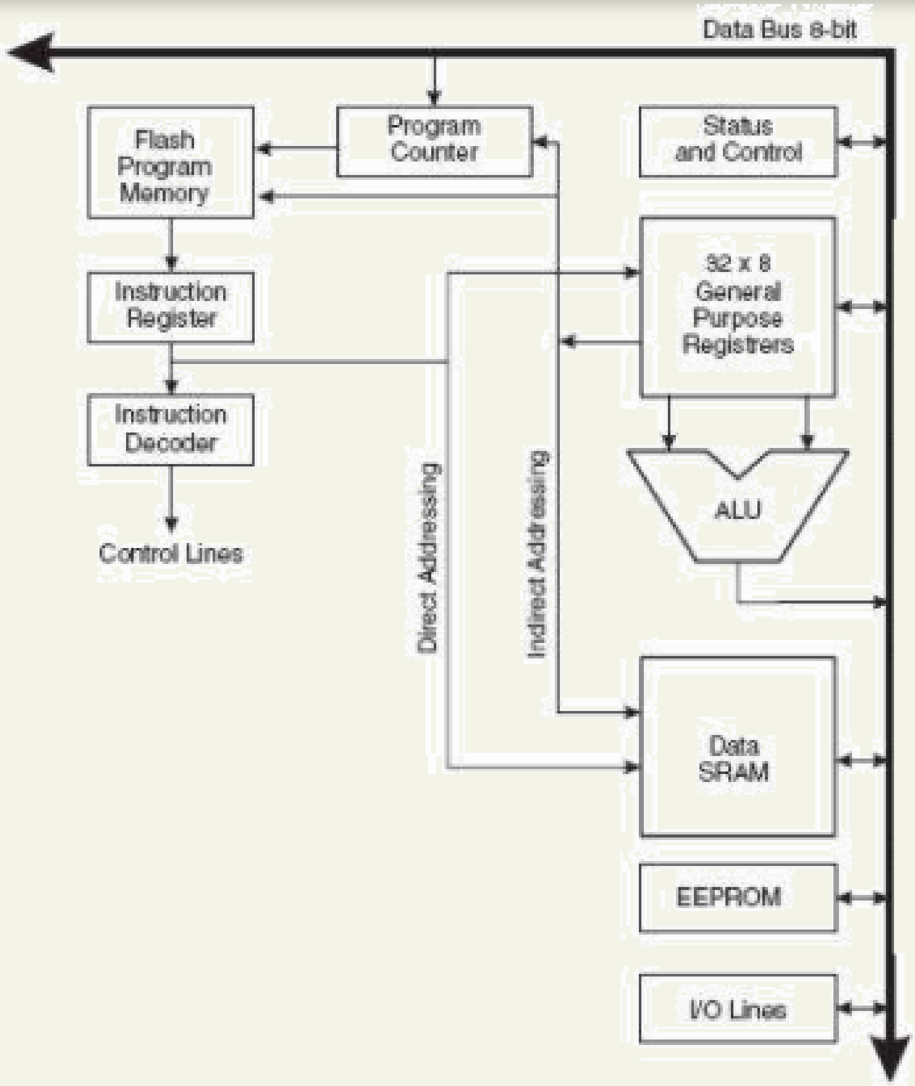
Refs: http://www.avr-asm-download.de/beginner_en.pdf (Gerhard Schmidt)

Why learn the ATMEGA128 Hardware Architecture?

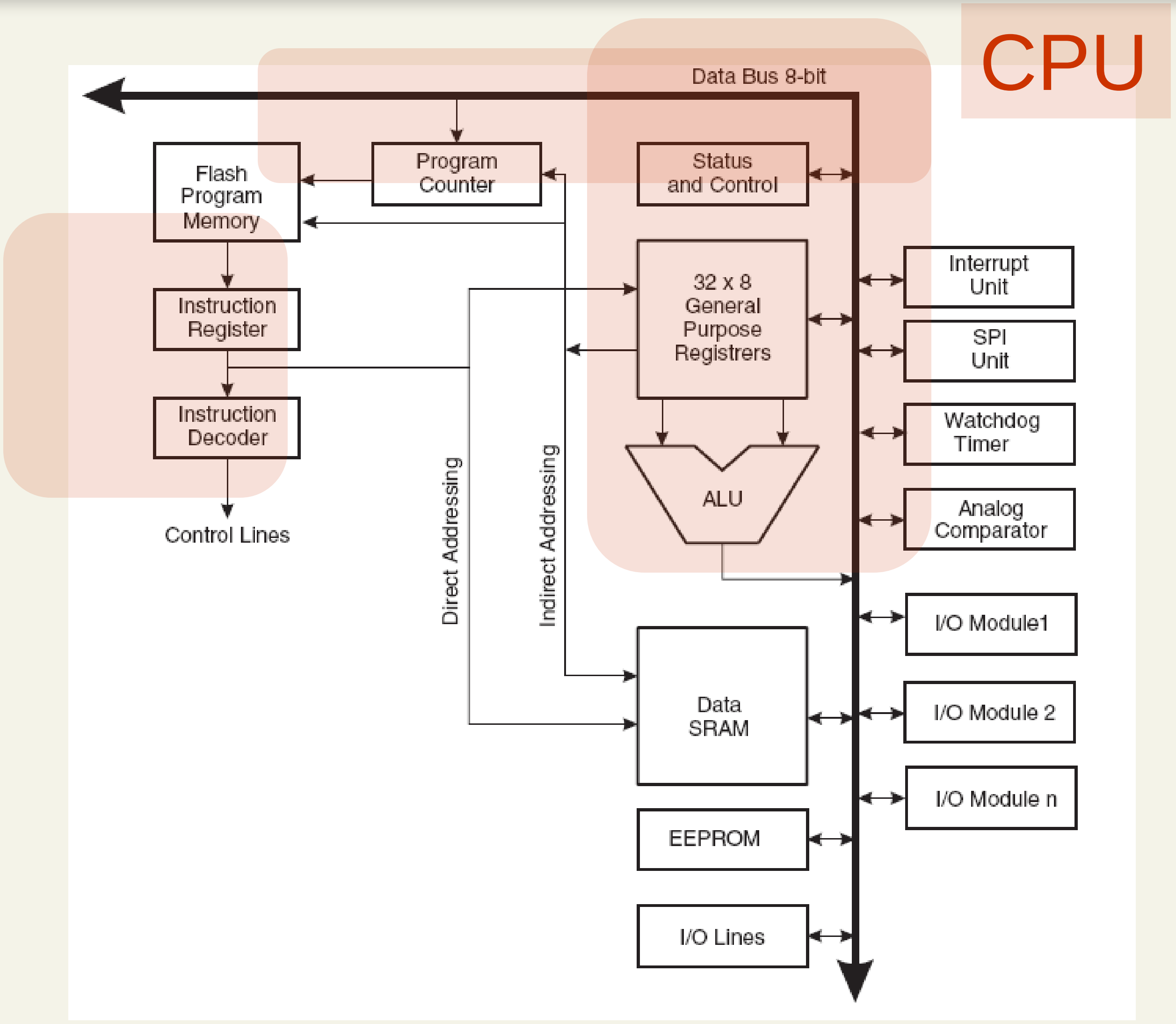
- Helps give intuition to why the assembly instructions were created the way the were
- Help understand what special feature may be available for you to make use of.

ATmega128 Architecture Overview

- 8 bit processor
  - size of bus is 8 bits
  - size of registers is 8 bits
- RISC architecture
- Harvard architecture
  - separate data and instruction memory
- 133 instructions



ATmega128 Block Diagram



ATMEGA128: RISC CPU Architecture

- What is RISC?
  - Reduced Instruction Set Computing (with respect to CISC, Complex Instruction set Computing)
- Typical RISC
  - LD/Store based: ALU to memory transaction via registers
  - Most instruction are the same length
  - Typically many less instructions than a CISC architecture
  - Typically many more registers than CISC since Data must be moved into a register before it can be operated on
  - Low number of instruction typically makes hardware design simpler (as compared to CISC)

Ref: http://www.seas.upenn.edu/~palsetia/cit595s07/RISCvsCISC.pdf (Diana Palsetia)

ATMEGA128: RISC vs CISC example

Size / time	**CISC**
1 byte, 1clk	**mov R1, 10**
1 byte, 1clk	**mov R2 5**
4 byte, 30 clk	**mul R2, R1**

Size / time	**RISC**
1 byte, 1clk	**mov R1, 0**
1 byte, 1clk	**mov R2, 10**
1 byte, 1 clk	**mov R3, 5**
1 byte, 1clk	**Begin: add R1, R2**
1 byte, 1 clk	**loop Begin**

- CISC: Instructions often variable length and variable time
- RISC: Instruction typically constant length and time
  - Almost all ATMEGA128 instructs take 1 clk (a few take 2 clks)
  - Simpler hardware logic for decoding instructions (thus typically faster)

Note: Many current RISC architectures do have a multiply (mul) instruction)

- ATMGA128: 2 clock cycles for integer multiply

Ref: http://www.seas.upenn.edu/~palsetia/cit595s07/RISCvsCISC.pdf (Diana Palsetia)

ATMEGA128 CPU Core Summary

Most instructions are 16-bit or 32-bit

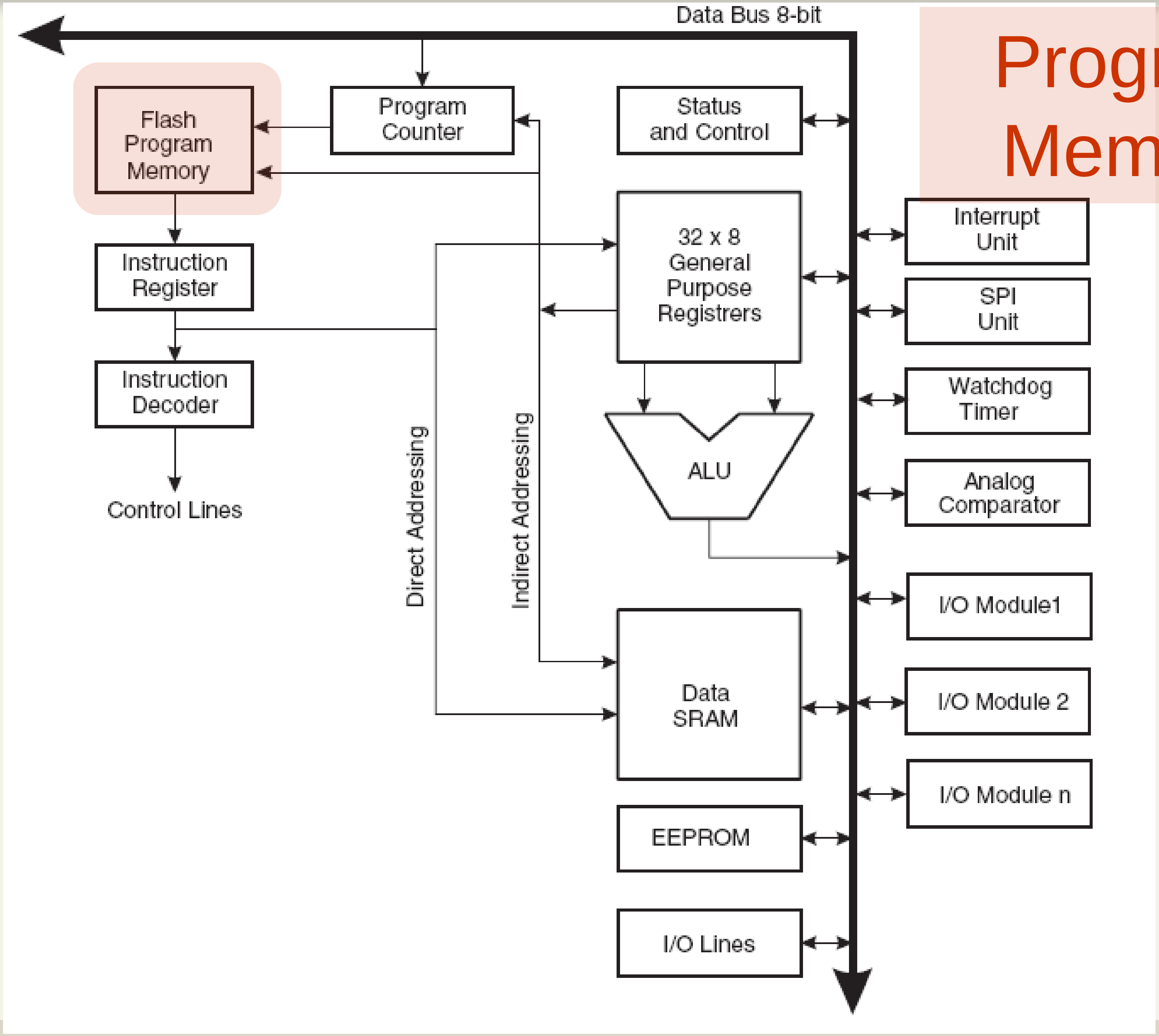
- Takes one or two cycles to fetch

Simple two-stage pipeline

- Most instructions take one or two cycles

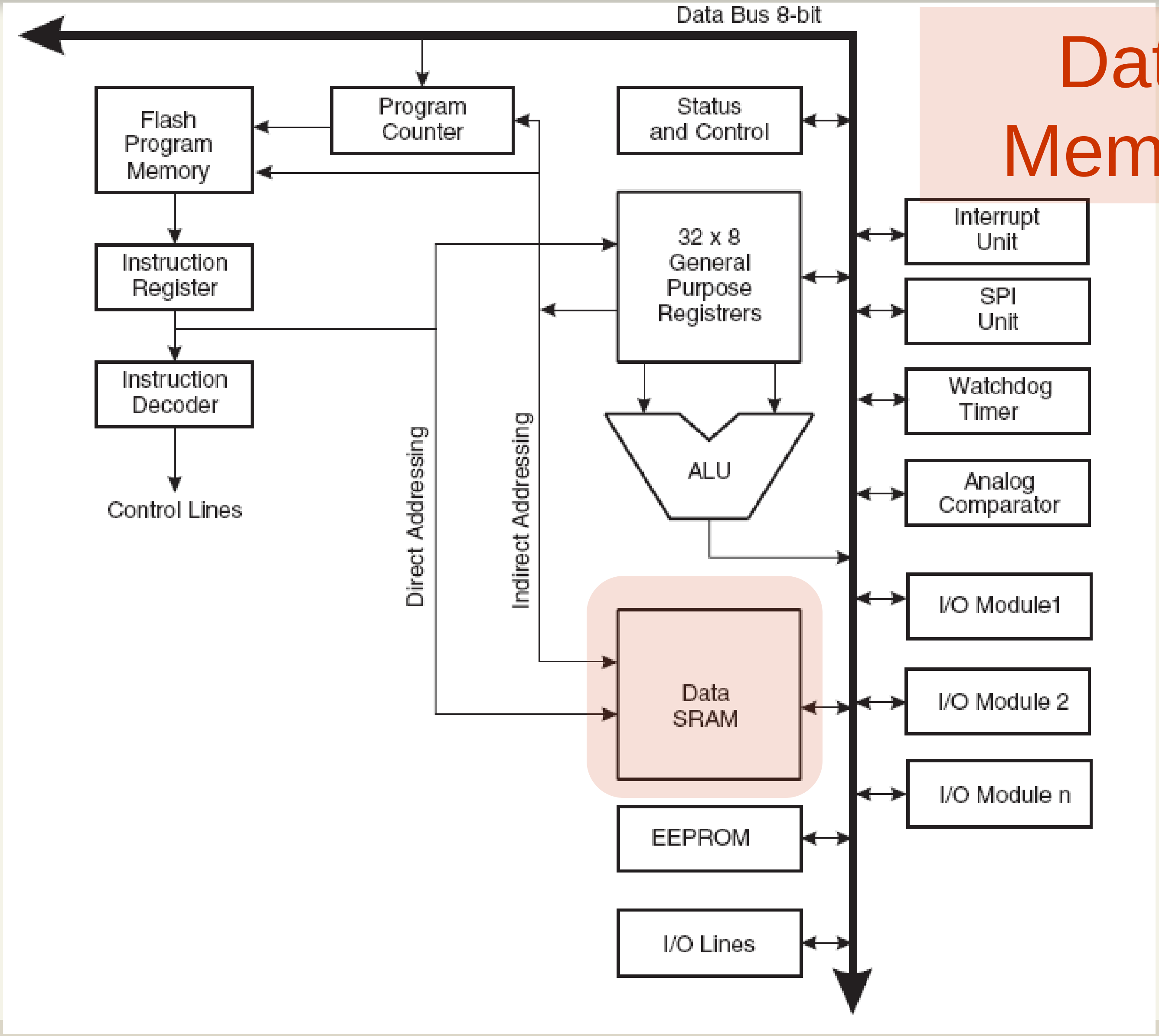
Registers are 8-bit and addresses are 16-bit

ATmega128 Block Diagram



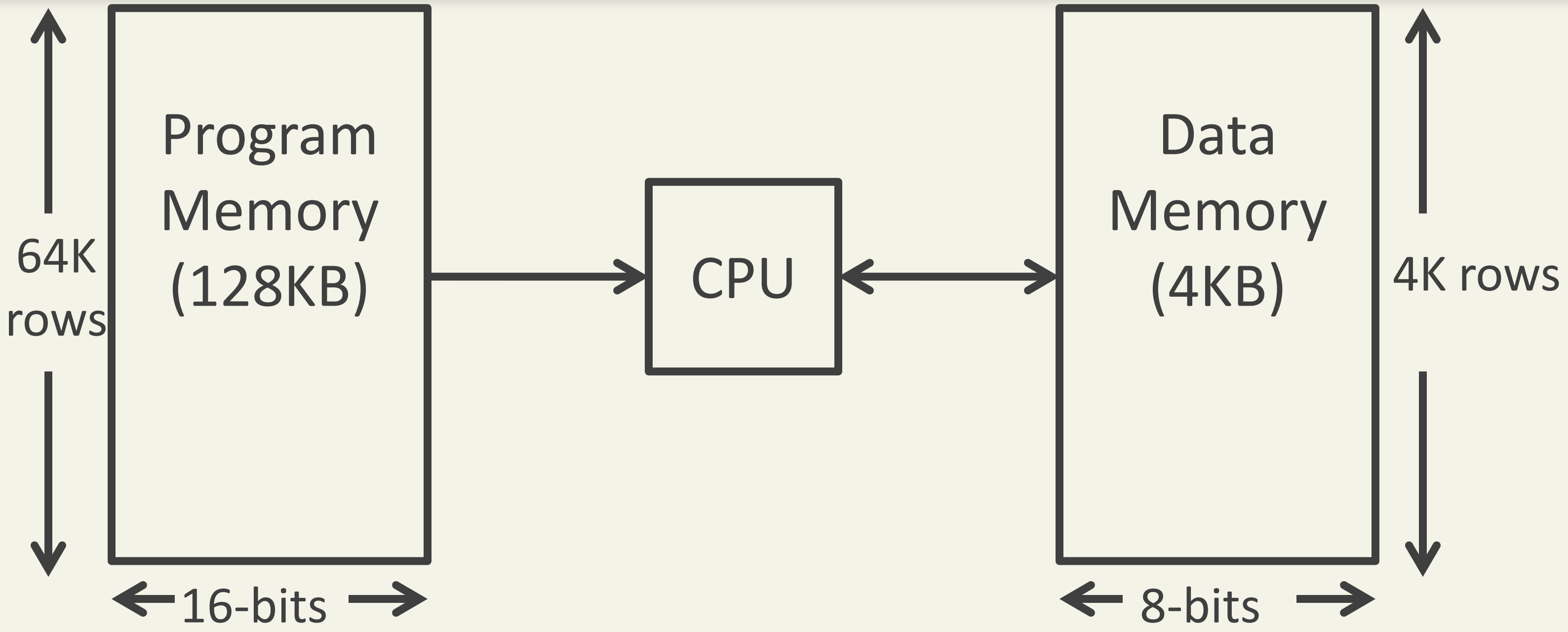
Program Memory

ATmega128 Block Diagram



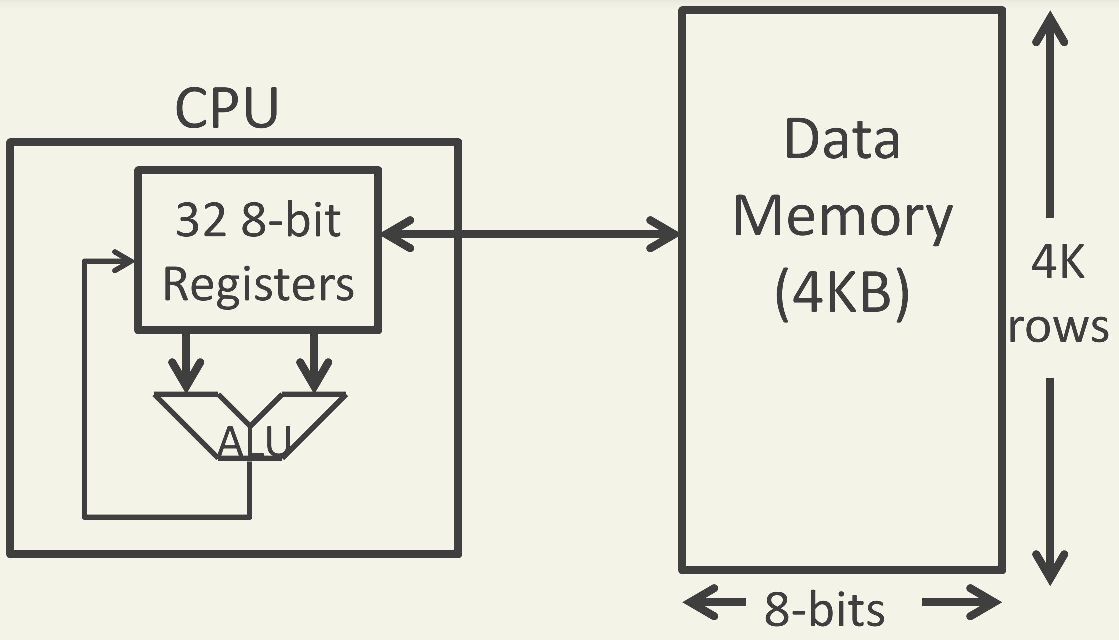
Data Memory

ATMEGA128: Harvard Architecture



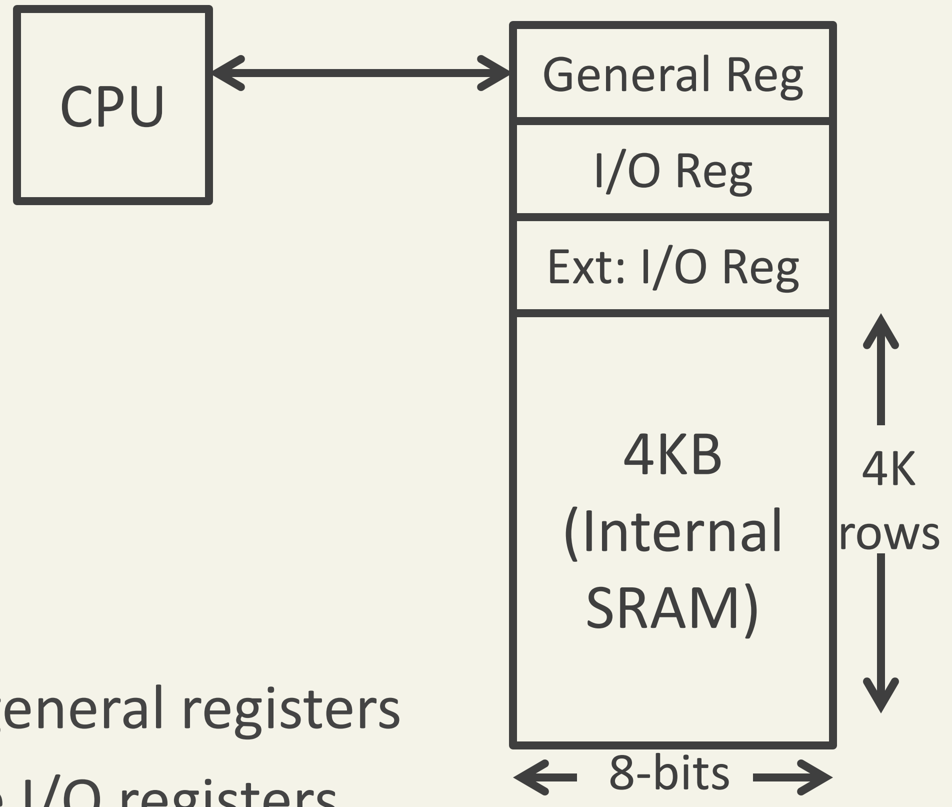
- Program memory
  - Flash based: Program stays even if power turned off (non-volatile)
  - 16-bits wide, instructions are 16-bit (typical) or 32-bit wide.
- Data Memory
  - SRAM based: Data disappears if power is turned off (volatile)
  - 8-bits wide: all data and registers are stored as 8-bit chunks.

ATMEGA128: Logical Data Memory organization



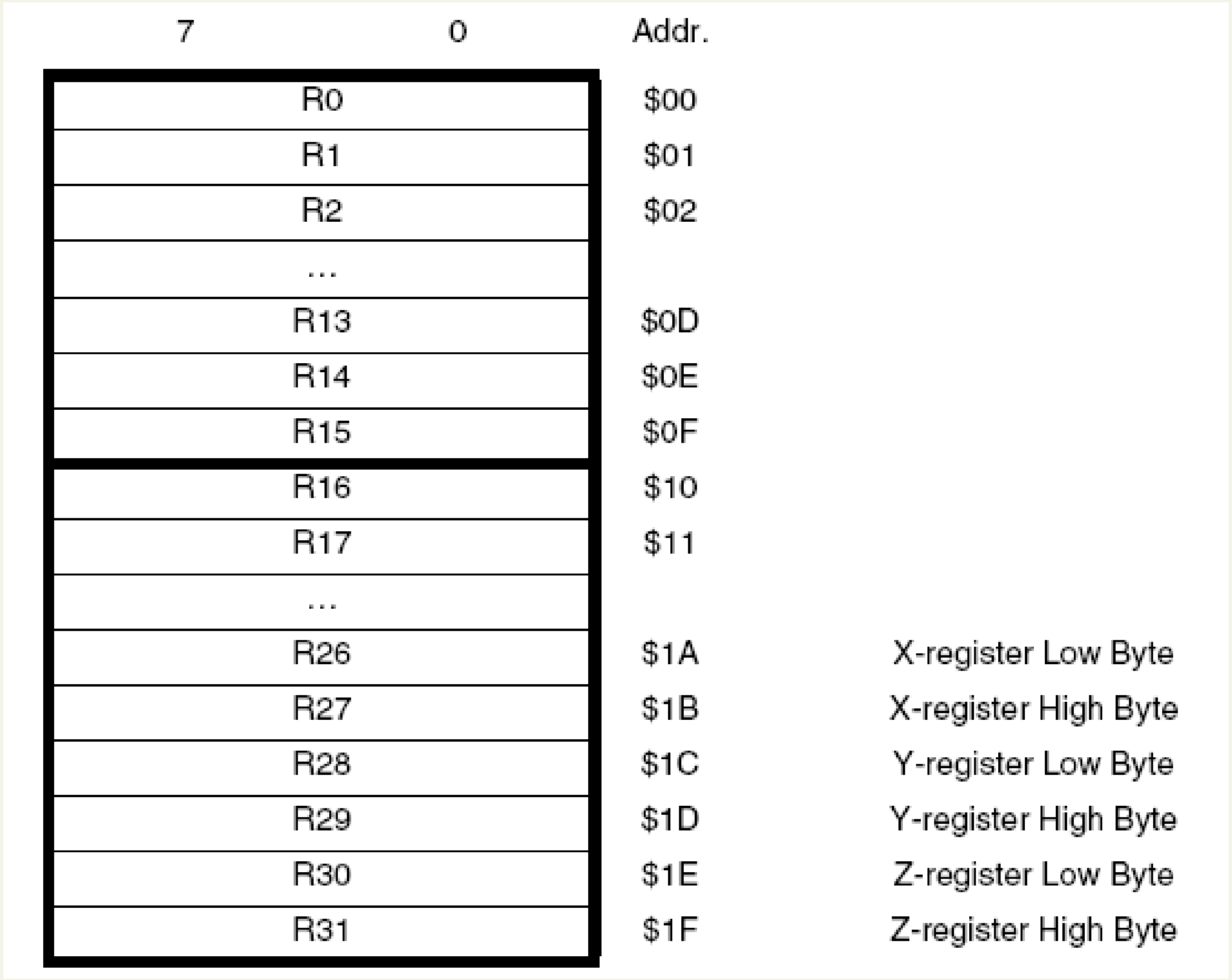
- Registers:
  - 8-bits wide
  - Directly accessible by ALU
- Data Memory:
  - 8-bit wide
  - Must use a register to move to from the ALU

ATMEGA128: Memory Map organization



- Address layout:
  - First 32 rows (0 – 0x1F) are general registers
  - Next 64 rows (0x20-0x5F) are I/O registers
  - Next 160 rows (0x60-0xFF) are Extend I/O registers
  - Next 4096 rows (0x0100 – 0x10FF) are Internal SRAM

General Purpose Register File



7 … 0	Addr.	
R0	$00	
R1	$01	
R2	$02	
…		
R13	$0D	
R14	$0E	
R15	$0F	
R16	$10	
R17	$11	
…		
R26	$1A	X-register Low Byte
R27	$1B	X-register High Byte
R28	$1C	Y-register Low Byte
R29	$1D	Y-register High Byte
R30	$1E	Z-register Low Byte
R31	$1F	Z-register High Byte

ATMEGA128: GP Registers (Cont.)

- 32 8-bit general purpose registers
  - Used for accessing SRAM
  - Used for storing function parameters
  - Used for instructions to execute operations on
- What is an 8-bit register.
  - Basically just 8 D-Flips connected together

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**

ATMEGA128: GP Registers (Cont.)

- R16 – R31 are the only registers that can have immediate values load to them (e.g. LDI R17, 5)

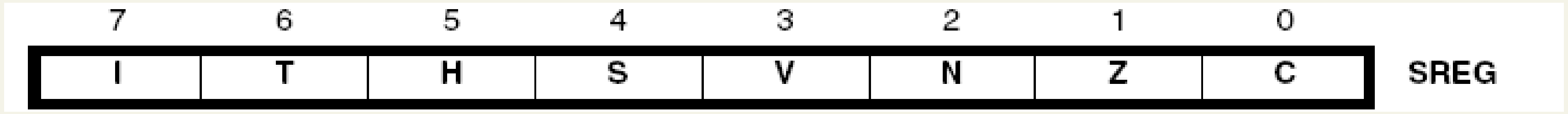
- Register pairs R27:R26, R29:R28, R31:R30 can be used as 16-bit pointers (short versions of these registers are X, Y, Z)

ATMEGA128: GP Registers

- 16-bit Datum stored across adjacent registers
  - Used for accessing SRAM
  - Used for storing function parameters
  - Used for instructions to execute operations on
- 32-bit stored across adjacent registers.

STATUS REGISTER (SREG)

Status Register (SREG)



7	6	5	4	3	2	1	0	
I	T	H	S	V	N	Z	C	SREG

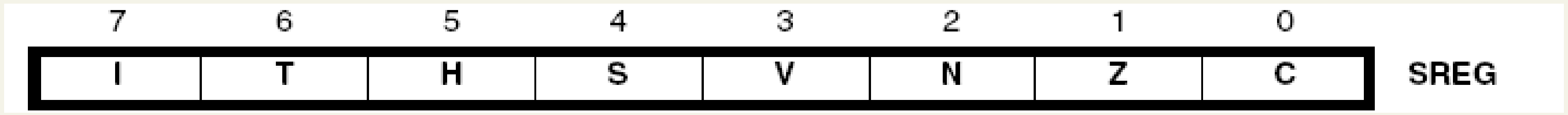
Describes the status of the CPU

I: Global Interrupt Enable, enable/disable interrupts to the CPU

T: Bit Copy Storage, for moving a single bit between registers

H: Half Carry Flag, To indicate a half carry in BCD arithmetic

Status Register (Cont.)



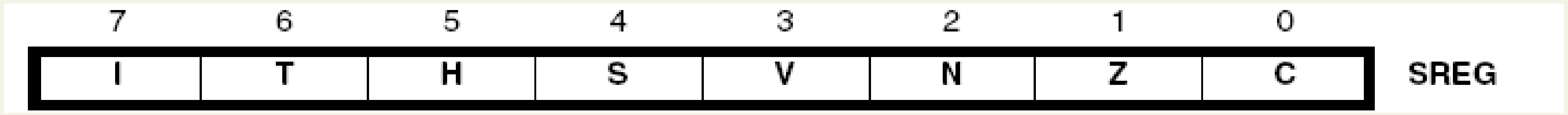
7	6	5	4	3	2	1	0	
I	T	H	S	V	N	Z	C	SREG

S: Sign Bit, Whether the ***actual*** result is negative or not with ***signed*** *type operation*

V: Two's Complement Overflow Flag, whether a overflow happened or not with *signed operands*

N: Negative Flag, whether the result is negative or not with *signed operands*

Status Register (Cont.)



Z: Zero flag, whether the result is zero or not

C: Carry Flag, whether a carry is generated for *unsigned operands*

H, S, N, V, Z, C bits are regarding the last arithmetic/logic operation

It is usually unknown if an instruction is working on signed or unsigned operations!

Designer's Perspective

H:	$Rd3 \bullet Rr3 + Rr3 \bullet \overline{R3} + \overline{R3} \bullet Rd3$ Set if there was a carry from bit 3; cleared otherwise
S:	$N \oplus V$, For signed tests.
V:	$Rd7 \bullet Rr7 \bullet \overline{R7} + \overline{Rd7} \bullet \overline{Rr7} \bullet R7$ Set if two's complement overflow resulted from the operation; cleared otherwise.
N:	$R7$ Set if MSB of the result is set; cleared otherwise.
Z:	$\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$ Set if the result is \$00; cleared otherwise.
C:	$Rd7 \bullet Rr7 + Rr7 \bullet \overline{R7} + \overline{R7} \bullet Rd7$ Set if there was carry from the MSB of the result; cleared otherwise.

Rd: The first operand register
Rr: The second operand register
R: The result register

Example: R7 refers to the 7th bit of the result.

Examples

- Let's look at some examples
  - See if you can guess the value of the H, S, V, N, Z, and C flags

Arithmetic and Logic Flags: Example

Add two operands: a + b

```
LDI   R24, 0x18          ; load a
LDI   R22, 0x09          ; load b
ADD   R24, R22           ; a+b
```

If a = 24, b = 9, what are the values for those flags?

a	b	result	Z	C	H	N	V	S
24	9	33	?	?	?	?	?	?

Arithmetic and Logic Flags: Example

a = **24** (0x18), b = **9** (0x09), result is 33 (0x21)

- Z = 0: Result **Not Zero**
  - The result 33 is not zero, no matter as signed or unsigned type
- N = 0: Result **Not Negative** as signed type
  - Look at the 7th bit of the result
  - The sign bit of 33 (0b$\underline{0}$0100001) is zero
- V = 0: Operation has **No Overflow** as signed type
  - 33 is a 7-bit value
- S = 0: **Actual result Not Negative** as signed type
  - The actual result is 33 for 24+9
  - $S = N \oplus V$
- C = 0: **No Carry**, or no overflow as unsigned type
  - The result 33 is an 8-bit value

Arithmetic and Logic Flags: Example

Add two operands: a + b

```
LDI   R24, 0x18          ; load a
LDI   R22, 0x09          ; load b
ADD   R24, R22           ; a+b
```

If a = 0x18, b = 0x09, what are the values for those flags?

a	b	result	Z	C	H	N	V	S
0x18	0x09	0x21	0	0	1	0	0	0

Arithmetic and Logic Flags: Example

a = **127** (0x7F), b = **127** (0x7F), result is 254 (0xFE) or -2

- Z = 0: Result **Not Zero**
  - The result -2 is not zero, no matter as signed or unsigned type
- **N = 1**: Result **Negative** as signed type
  - The sign bit of 254 (0b<u>1</u>1111110) is one
- **V = 1**: Result has **Overflow** as signed type
  - 254 is not a 7-bit value, it becomes -2
- S = 0: **Actual result Not Negative** as signed type
  - The actual result 254 is positive; overflow changed the sign type
  - $S = N \oplus V = 1 \oplus 1 = 0$
- C = 0: **No Carry**
  - The result 254 is an 8-bit value

Arithmetic and Logic Flags: Example

a = **1** (0x01), b = **255** or **-1** (0xFF), result is 0 (0x00)

- **Z = 1**: Result **Zero**
  - No matter as signed or unsigned type
- N = 0: Result **Not Negative** as signed type
  - The sign bit of 0 (0b<u>0</u>0000000) is zero
- V = 0: Result has no **Overflow** as signed type
  - 0 is the actual result
- S = 0: **Actual result Not Negative** as signed type
  - 0 is not negative
  - $S = N \oplus V = 1 \oplus 1 = 0$
- **C = 1**: **Carry** occurs, operation overflows as unsigned type
  - 1+255 = 256 is not an 8-bit value, it becomes 0

Arithmetic and Logic Flags: Example

How do we know if a result is positive as **unsigned** type?

```
LDI  R24, 2            ; load a
LDI  R22, 1            ; load b
SUB  R24, R22          ; a-b
```

If a = 2, b = 1, what are the values for those flags?

- Two’s complement: 2 – 1 = 2 + (-1) = **0x02 + 0xFF** = 0x01
- Z=0, N=0, V=0, S=0
- **C=1**, the result is positive as unsigned type

If a = 1, b = 2

- Two’s complement: 1 – 2 = 1 + (-2) = 0x01 + 0x FE = 0xFF
- **C = 0**, the result is not positive as unsigned type

Stack Pointer (SP)

The Stack Pointer Register: pointing to the top of the stack.

- Stack pointer is implemented as two 8-bit **I/O registers**
- SPH:SPL (most significant: lest significant byte)
- Example for setting the SP register to top of stack

```
.DEF MyPreferredRegister = R16
.DEF RAMEND = $10FF
LDI MyPreferredRegister, HIGH(RAMEND) ; Upper byte
OUT SPH,MyPreferredRegister ; to stack pointer
LDI MyPreferredRegister, LOW(RAMEND) ; Lower byte
OUT SPL,MyPreferredRegister ; to stack pointer
```

Stack Pointer (SP) (Cont.)

Using Stack Pointer Register

- Place value onto the stack (Push)
  - Remember Stack starts at the highest address and grows downward. Thus "push" decrements SP.

PUSH R16 ; Throw that value in R16 on top of the stack

- Remove value from the stack (Pop). "pop" increments SP, i.e. makes the stake smaller

POP R16 ; Read value from the top of the stack, place in R16

Special Purpose Registers

Extension to General Purpose Registers

- Stack pointer
- Ports A, B, C, D, E, F, G, …
- Registers related to interrupt
- And more …

Example of difference: only GPRs in

ADD R*x*, R*y*

ATmega128 I/O Ports

I/O port registers are in the I/O spaces

- They also have their own memory addresses

```
/* Input Pins, Port E */
#define PINE      _SFR_IO8(0x01)

/* Data Direction Register, Port E */
#define DDRE      _SFR_IO8(0x02)

/* Data Register, Port E */
#define PORTE     _SFR_IO8(0x03)
```

Summary of AVR Registers

1. **GPRs: R0-R31**
   - R26/R27, R28/R29, R30/R31 are **X, Y, Z registers**
2. Status register **SREG**
   - **H, S, N, V, Z, C** bits
3. Stack pointer **SP**
4. Special purpose registers **SPRs**
   - SP is a SPR

ATmega128 Memory Address

- GPRs R0-R31: addresses 0x0000-0x001F
  – Directly accessed by ALU instructions and by memory instructions
- I/O registers (space): 0x0020-0x005F
  – Directly accessed by IN/OUT instructions and by memory instructions
- Extended I/O registers: 0x0060-0x00FF
  – Directly accessed by memory instructions only
- Normal memory: 0x0100 above
  – Directly accessed by memory instructions only

Example AVR Assembly

```
int a;
a = a + 10;
```

```
LDS  R24, a    ; Load a's lower 8-bit
LDS  R25, a+1  ; Load a's upper 8-bit
ADIW R24, 10   ; R24/R25 ←R24/R25+10
STS  a, R24    ; save a's upper half
STS  a+1, R25  ; save a's lower half
```

Example AVR Assembly

```
if (a > 0)
   …

CLR  R1          ; R1 ← 0
CP   R1, R24     ; compare lower half
CPC  R1, R25     ; compare higher half
BRGE else1       ; branch if greater than
                 ; or equal
```

How to Study Assembly

1. Get to know CPU registers

2. Know basic types of Instruction
   - Memory load and store
   - Arithmetic/Logic
   - Compare and branch

How to Study Assembly

3. Translate C statements
   - Memory accesses
   - Simple arithmetic statements
   - If statement
   - Loop statements
4. Translate C functions
   - Function Linkage
   - Making a function call

How to Study Assembly

5. Interrupt System

   Principle of interrupt and exception

   Interrupt vector table

   Saving and restoring context

Challenges

Challenges in learning Assembly

- Must understand how CPU works cycle by cycle
- Have to memorize some notations before fully understanding them