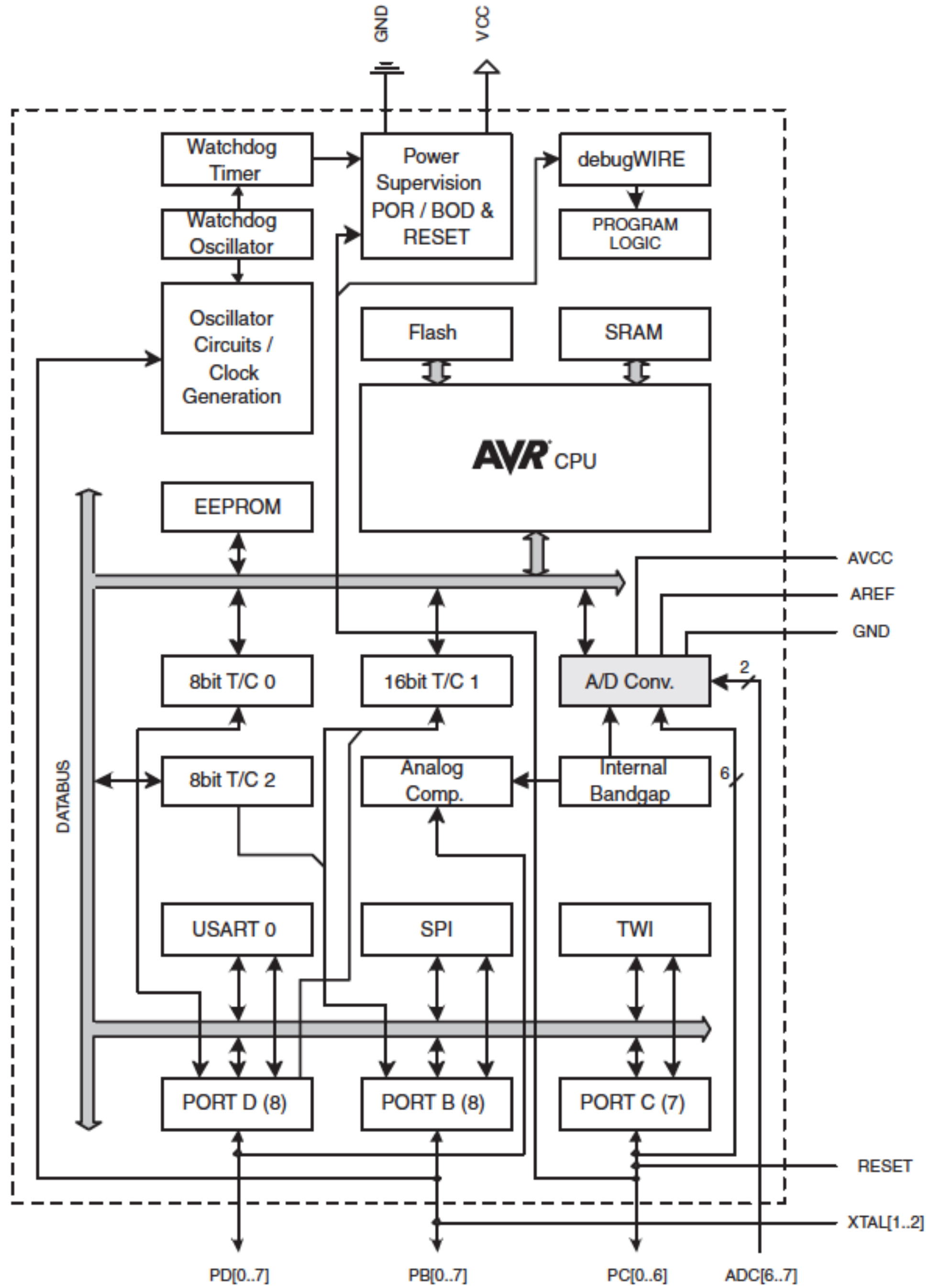
Αρχιτεκτονική των μικροελεγκτών AVR

Δρ Ασημάκης Λερός, Τμήμα ΜΠΕΣ

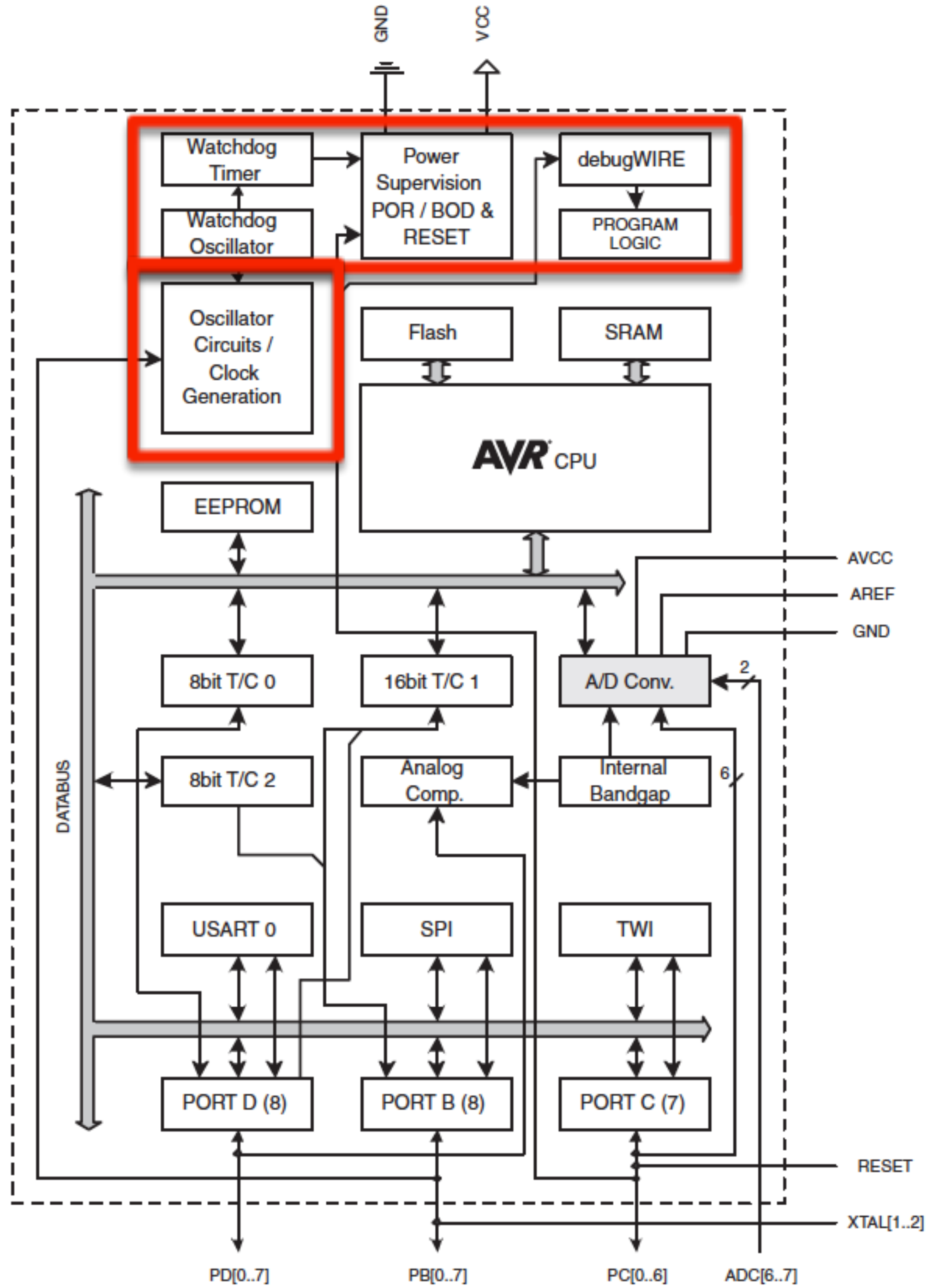
Το υλικό είναι από τα datasheets του ATMega328 και το βιβλίο των Harris & Harris (βιβλιοθήκη)

AVR Architecture



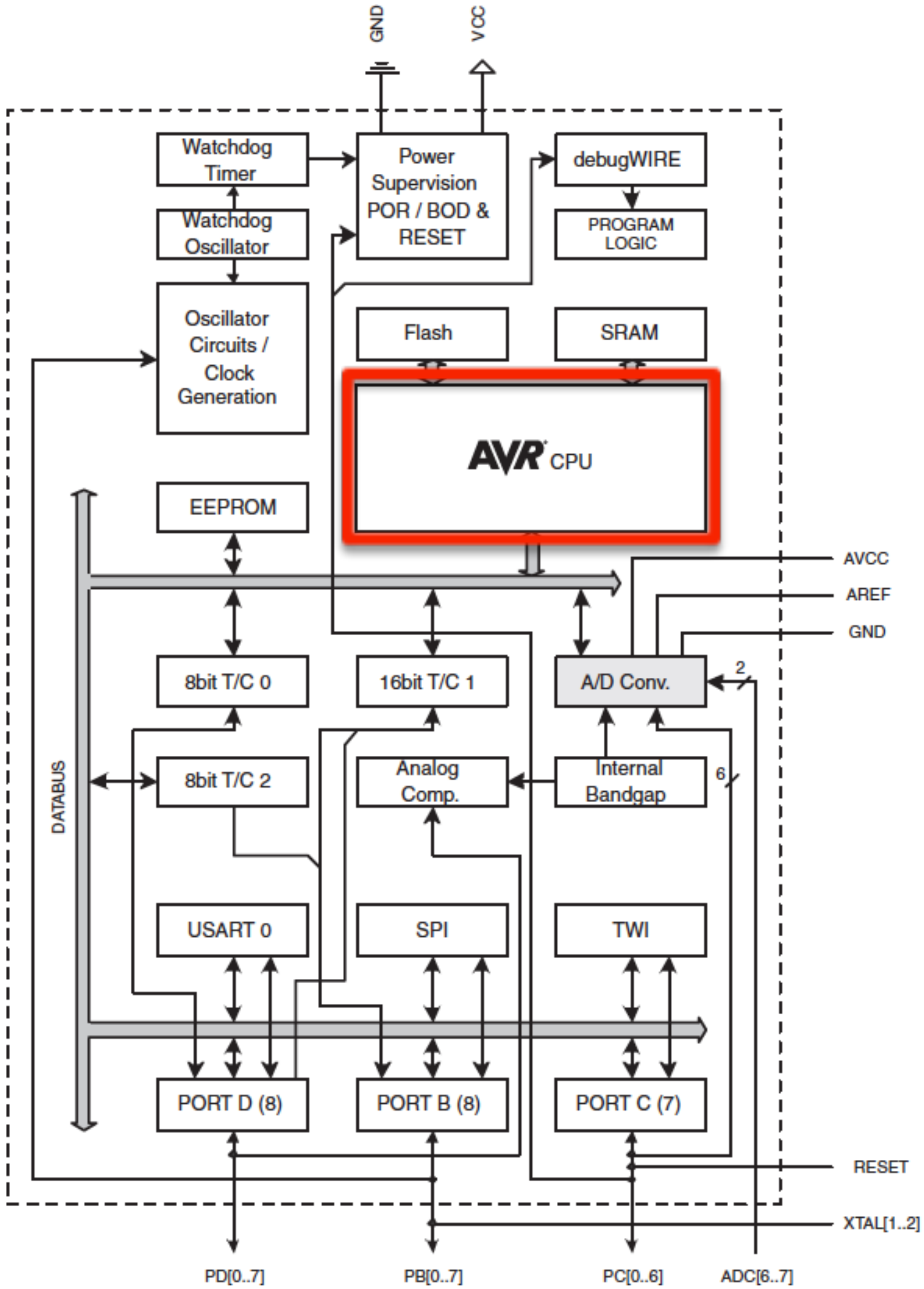
AVR Architecture

- Clocks and Power
- Beyond scope of this course



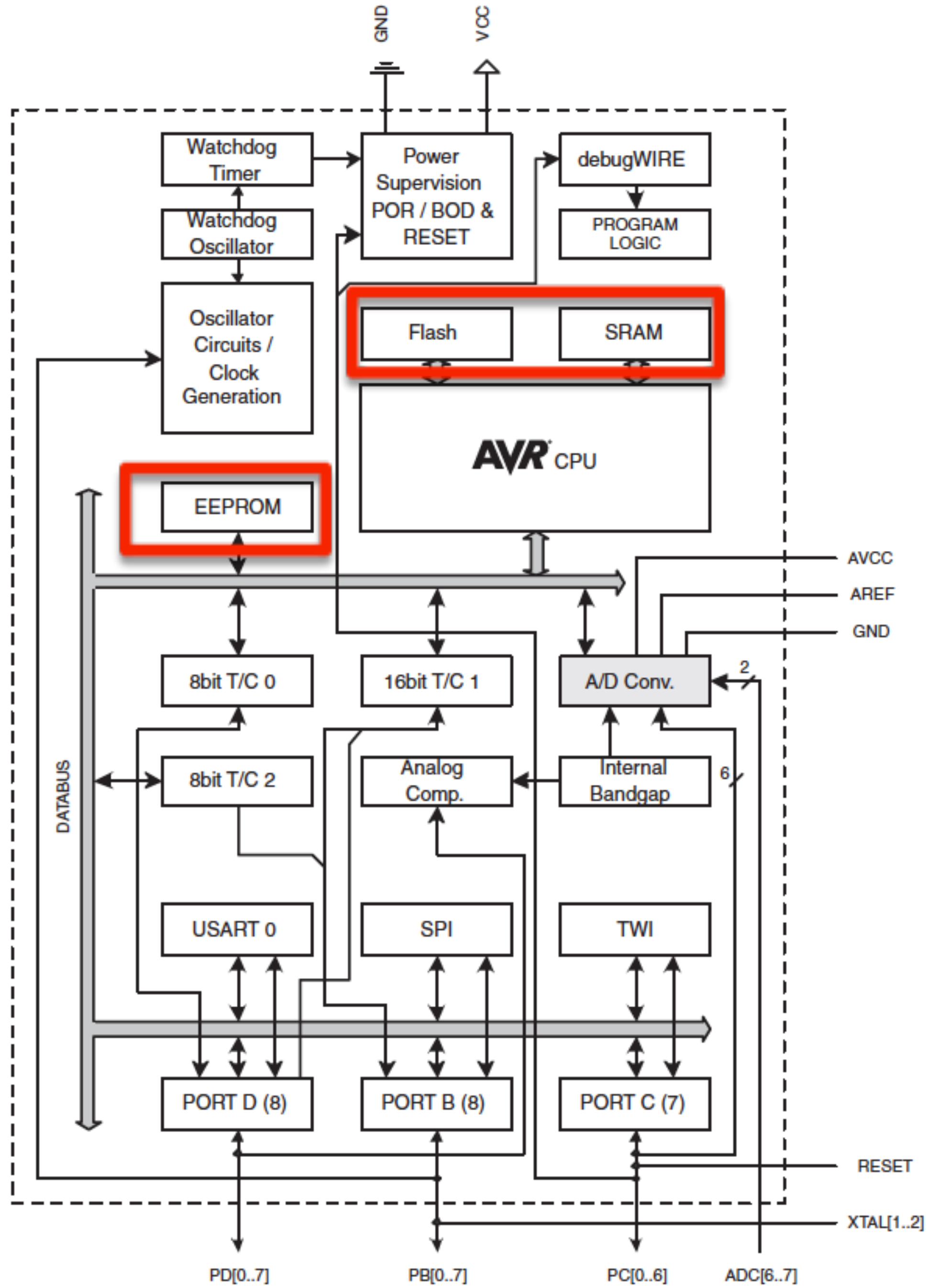
AVR Architecture

- CPU
  - Details coming



AVR Architecture

- Harvard architecture
- Flash – program memory
  - 32K
- SRAM – data memory
  - 2K
- EEPROM
  - For long-term data
  - On I/O data bus

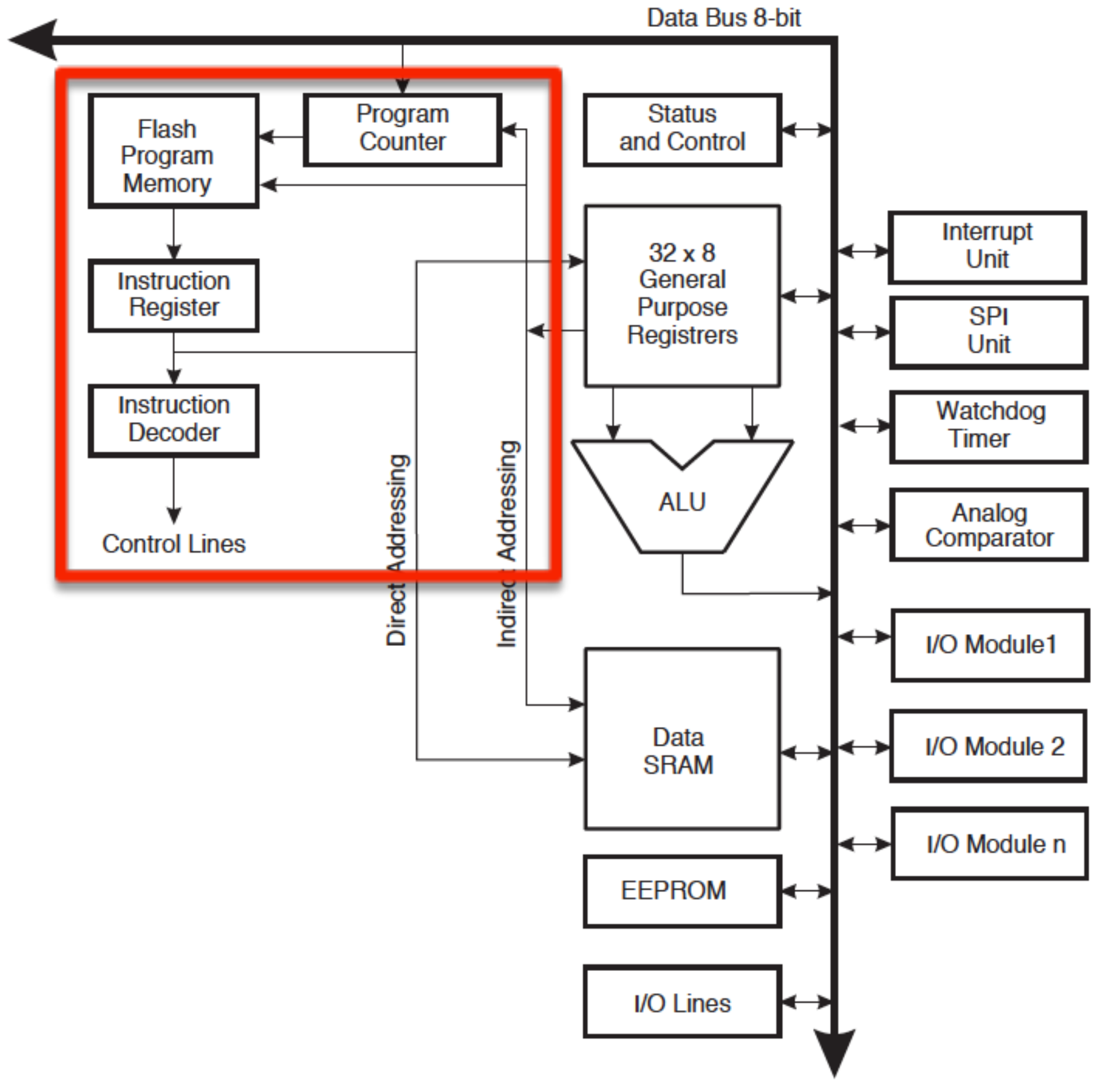


Memory

- Flash (32K) (15-bit addresses)
  - Program memory – read only
  - Non-volatile
  - Allocate data to Flash using PROGMEM keyword
    - see documentation
- SRAM (2K)
  - Temporary values, stack, etc.
  - Volatile
  - Limited space!
- EEPROM (1K)
  - Long-term data
  - see documentation on EEPROM library

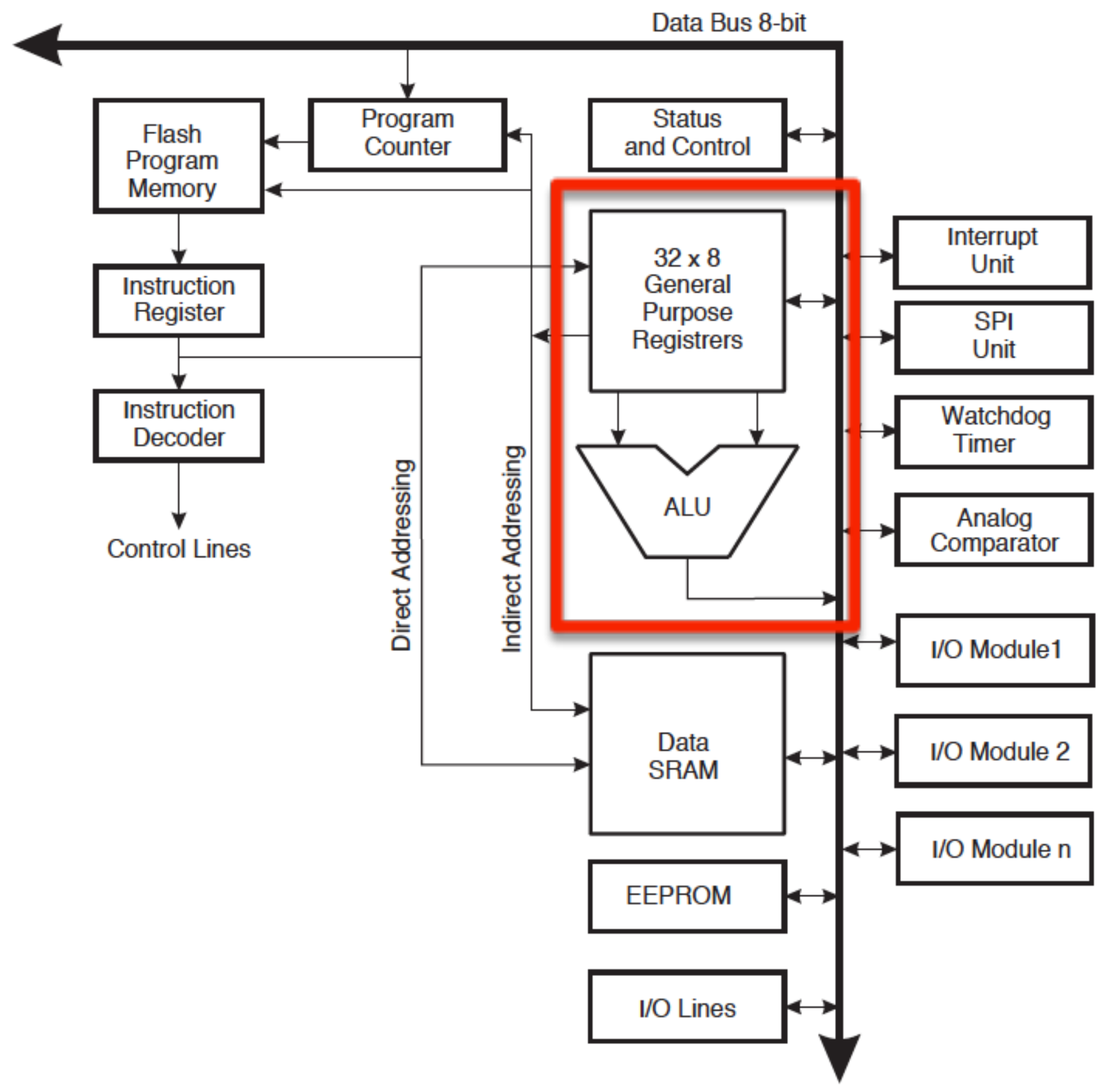
AVR CPU

- Instruction Fetch and Decode



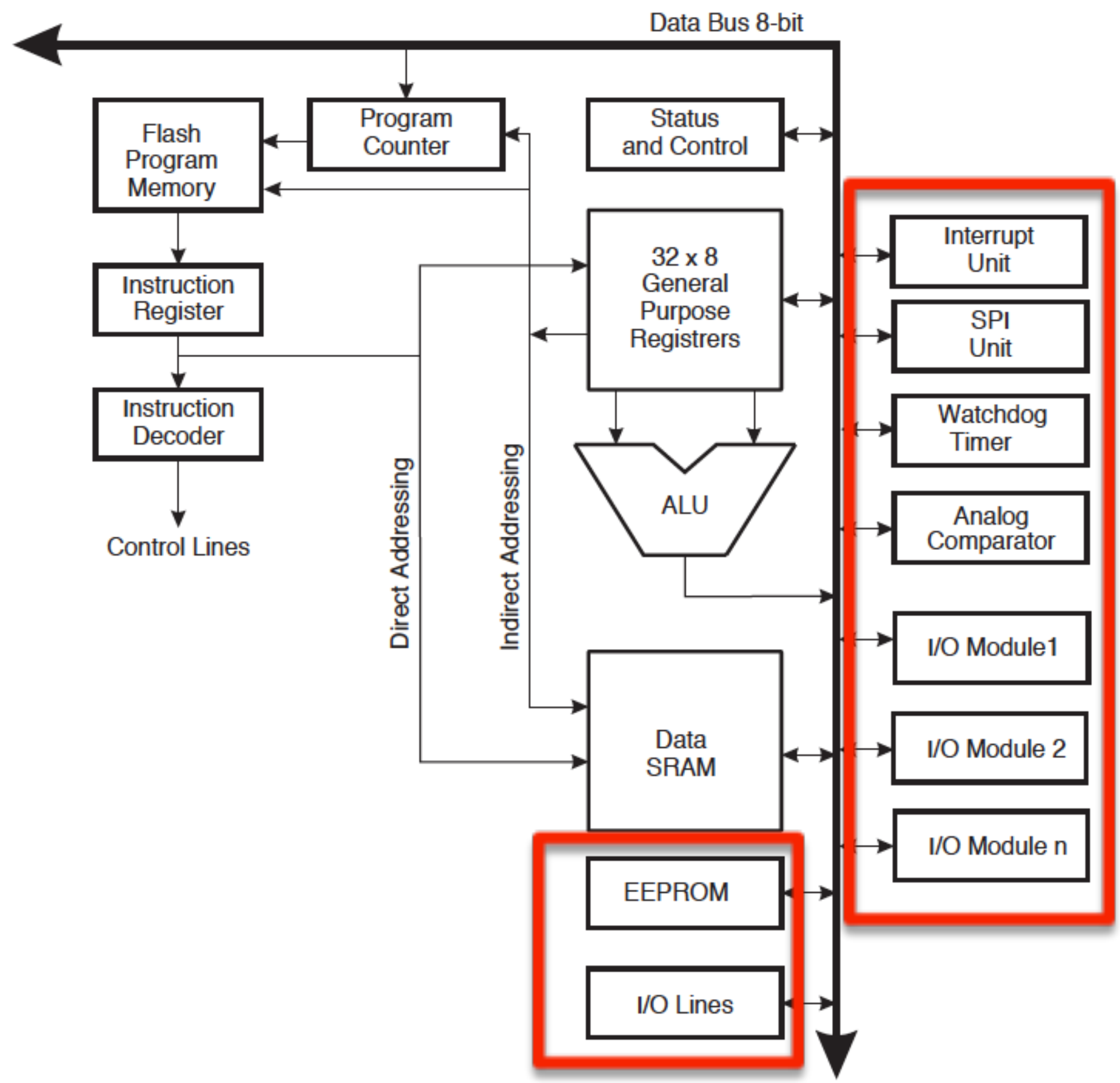
AVR CPU

- ALU Instructions



AVR CPU

- I/O and special functions



AVR Register File

- 32 8-bit GP registers
- Part of SRAM memory space

Figure 6-2. AVR CPU General Purpose Working Registers

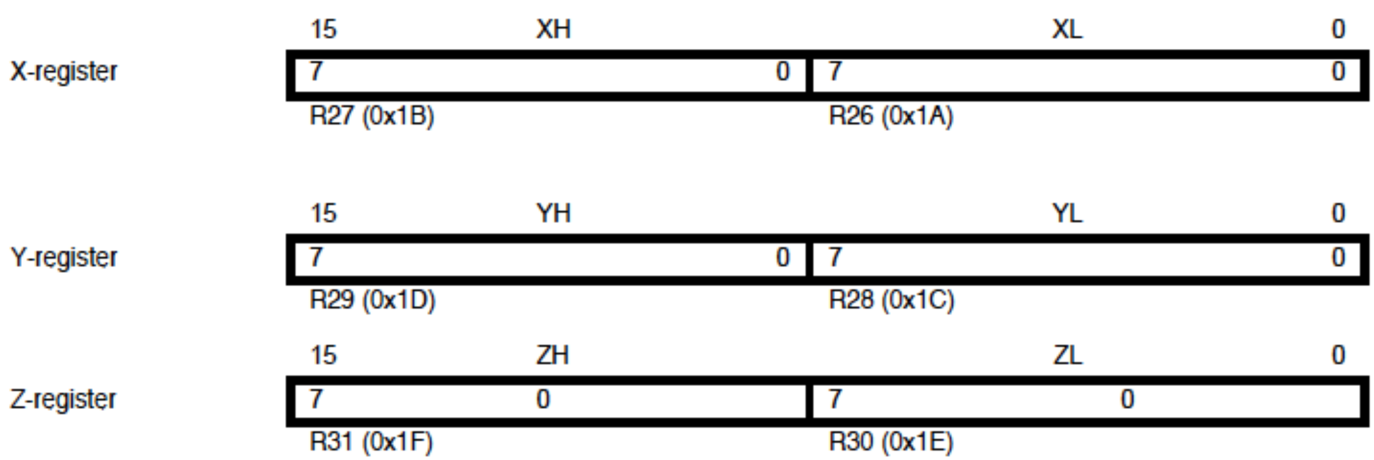
General Purpose Working Registers

7 … 0	Addr.	
R0	0x00	
R1	0x01	
R2	0x02	
...		
R13	0x0D	
R14	0x0E	
R15	0x0F	
R16	0x10	
R17	0x11	
...		
R26	0x1A	X-register Low Byte
R27	0x1B	X-register High Byte
R28	0x1C	Y-register Low Byte
R29	0x1D	Y-register High Byte
R30	0x1E	Z-register Low Byte
R31	0x1F	Z-register High Byte

Special Addressing Registers

- X,Y and Z registers
  - 16-bit registers made using registers 26 – 31
- Support indirect addressing

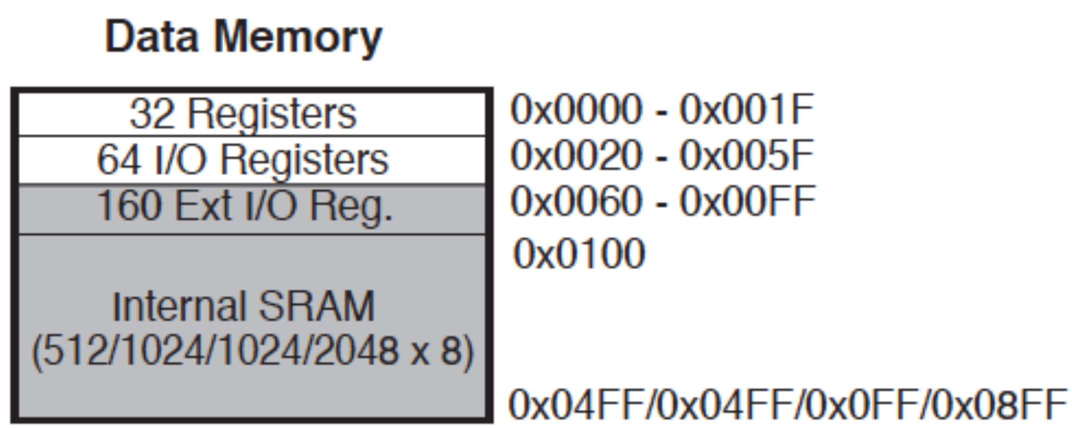
Figure 6-3. The X-, Y-, and Z-registers



AVR Memory

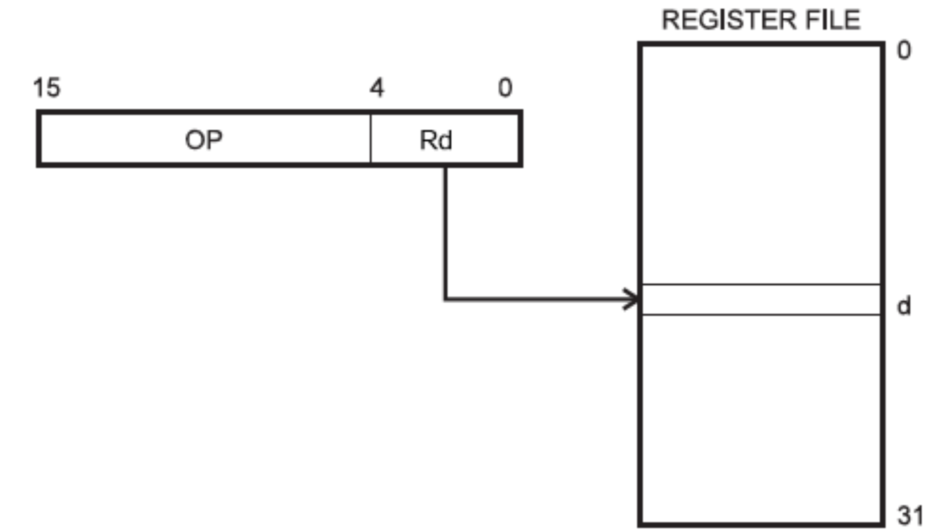
- Program memory – Flash

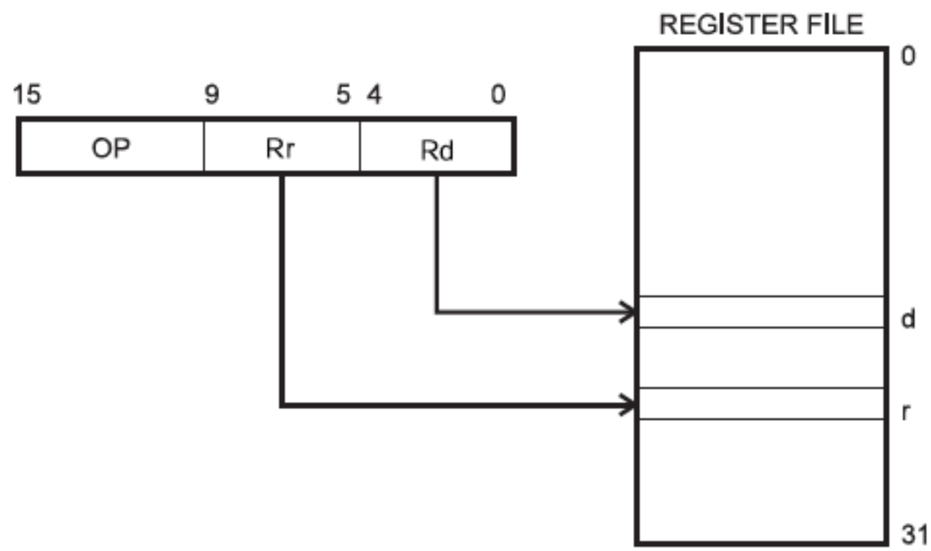
- Data memory - SRAM



Addressing Modes

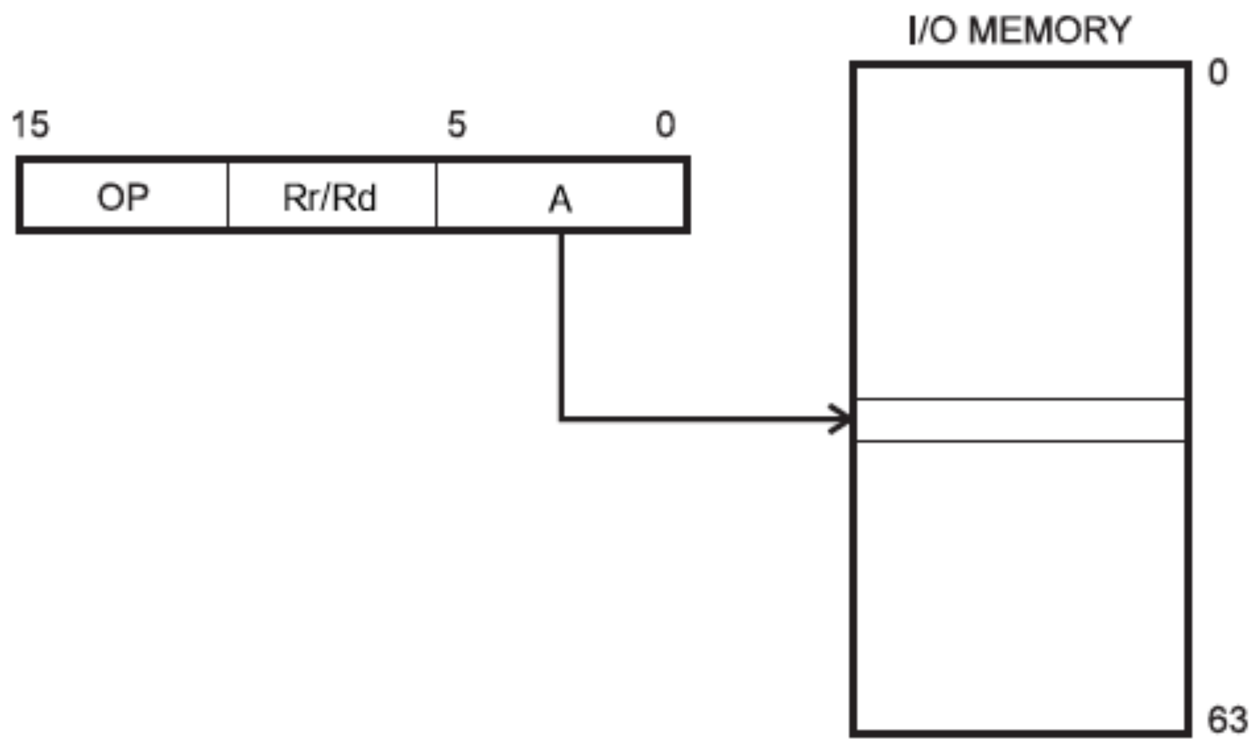
- Direct register addressing





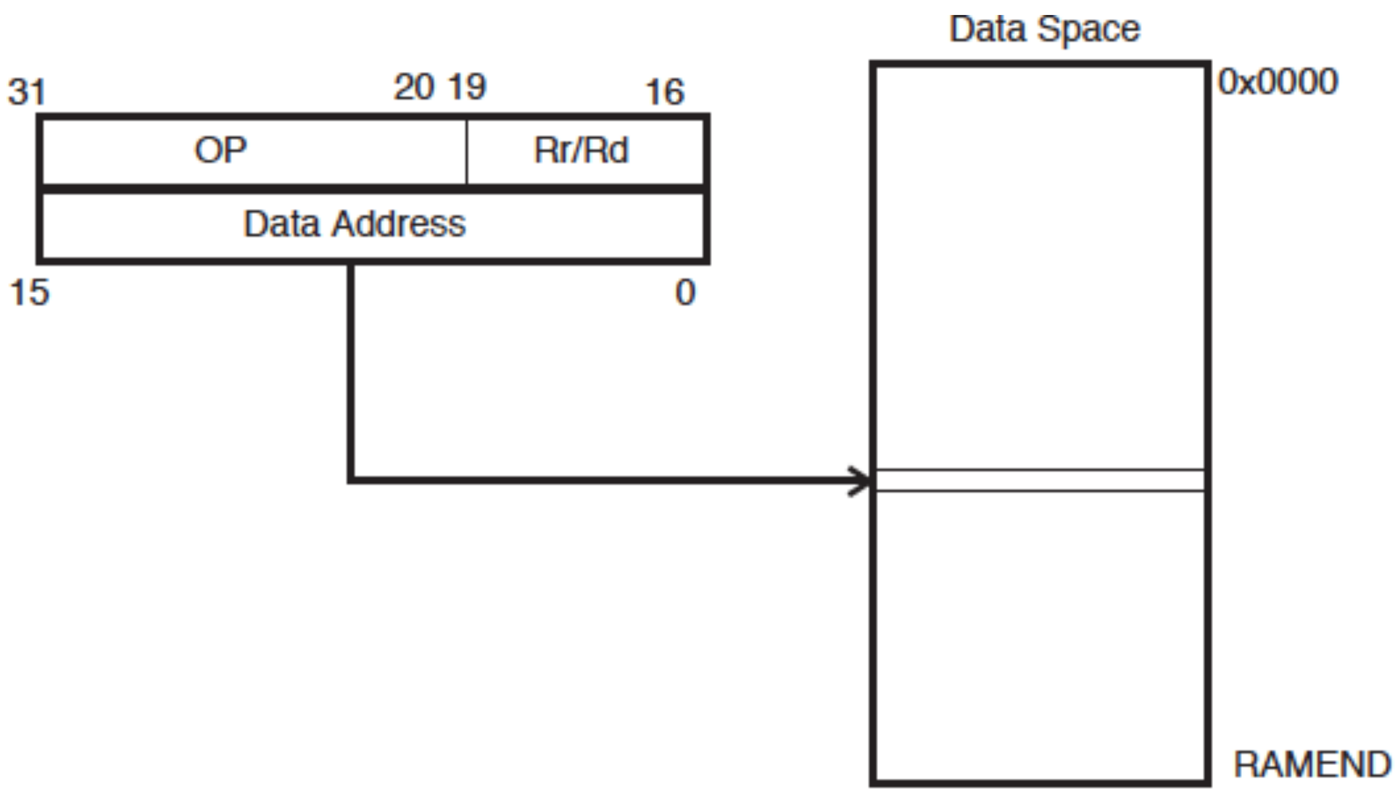
Addressing Modes

- Direct I/O addressing



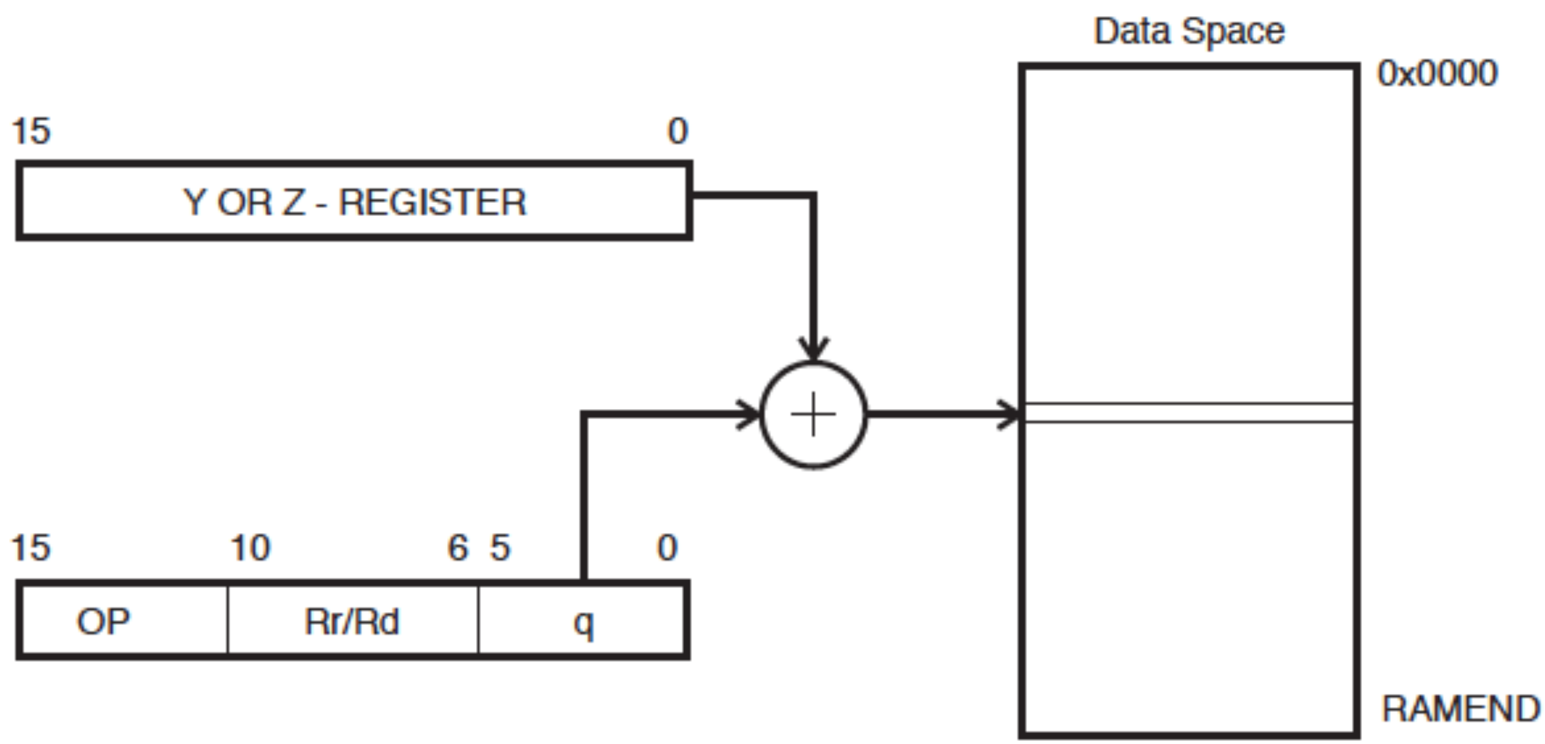
Addressing Modes

- Direct data memory addressing



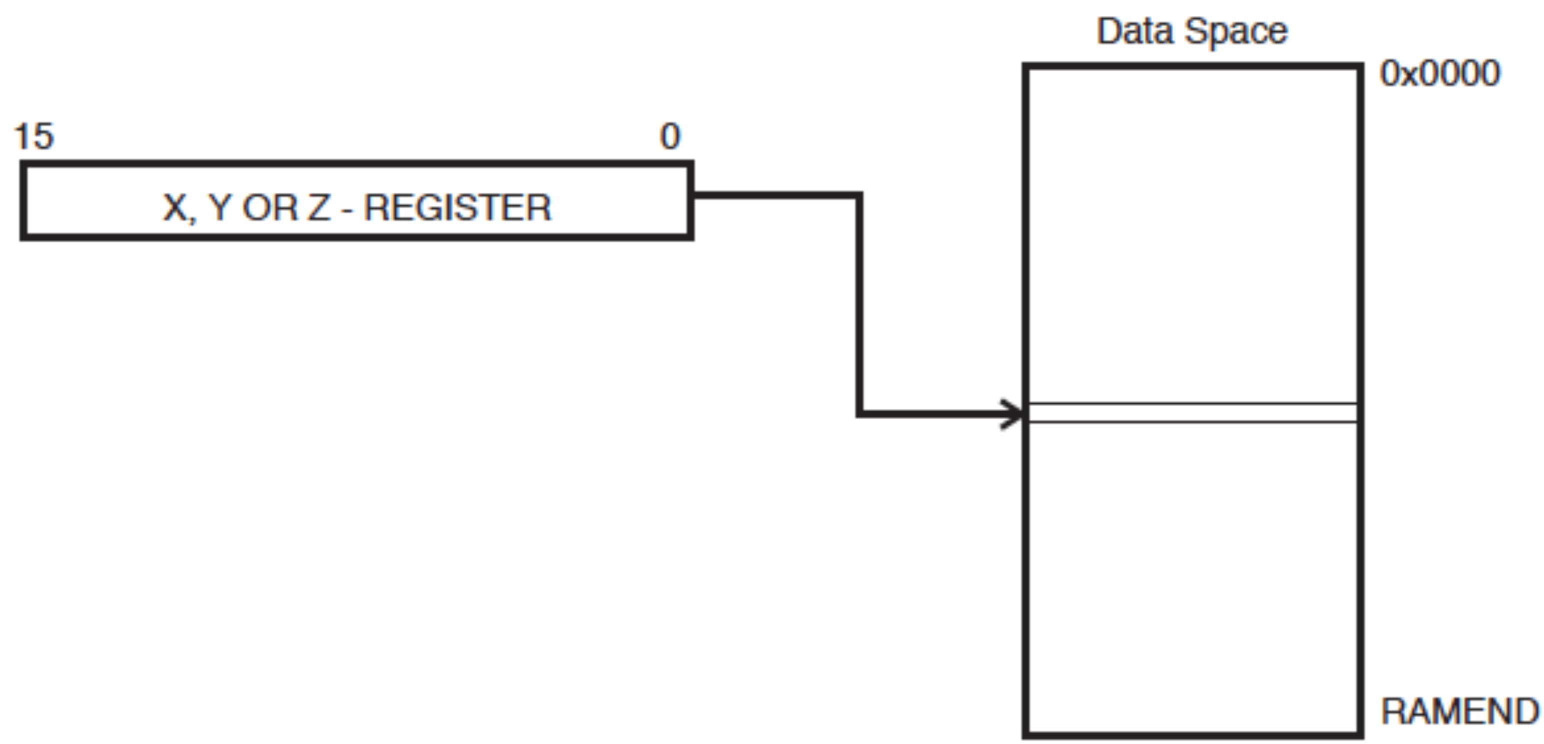
Addressing Modes

- Direct data memory with displacement addressing



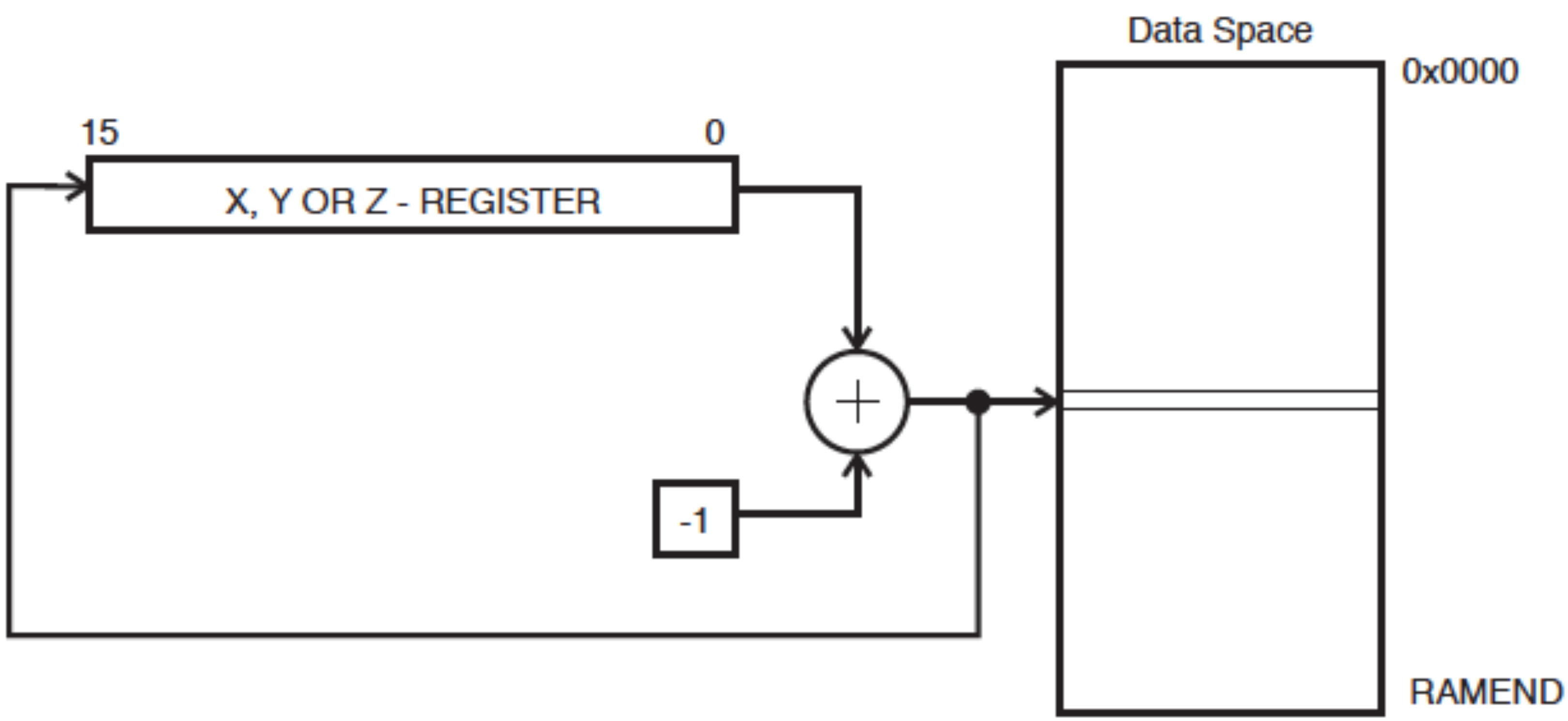
Addressing Modes

- Indirect data memory addressing



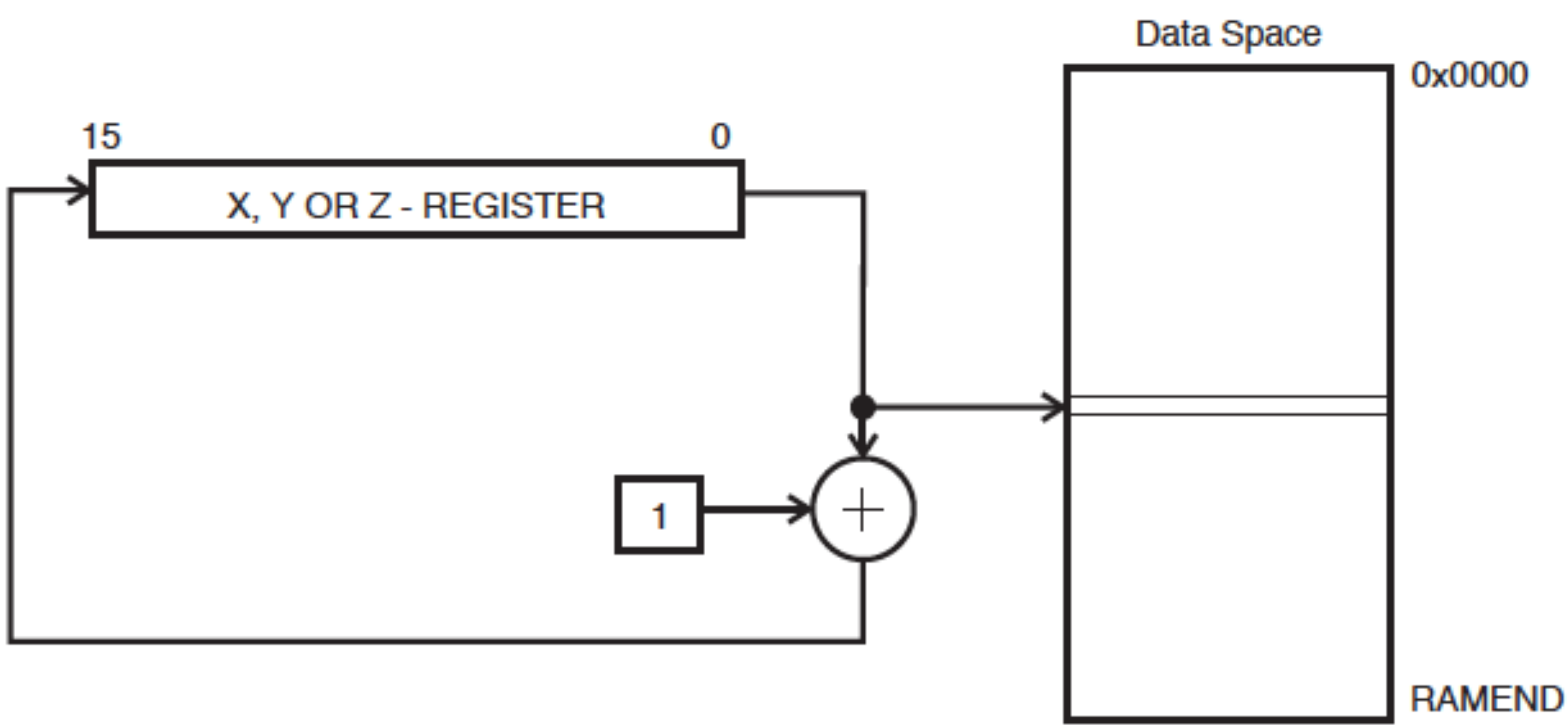
Addressing Modes

- Indirect data memory addressing with pre-decrement



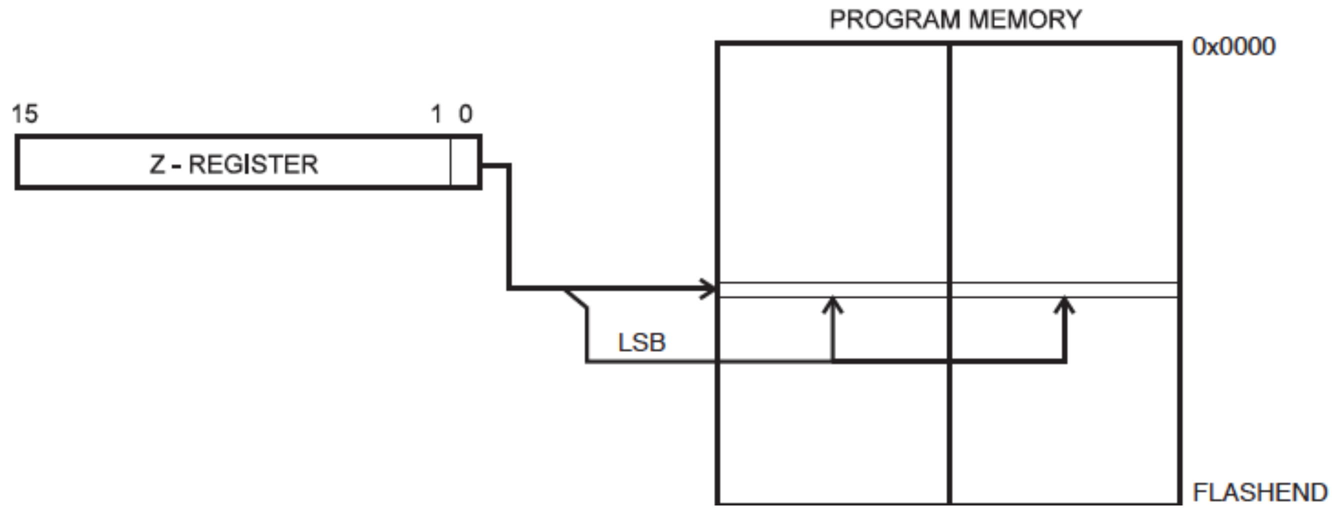
Addressing Modes

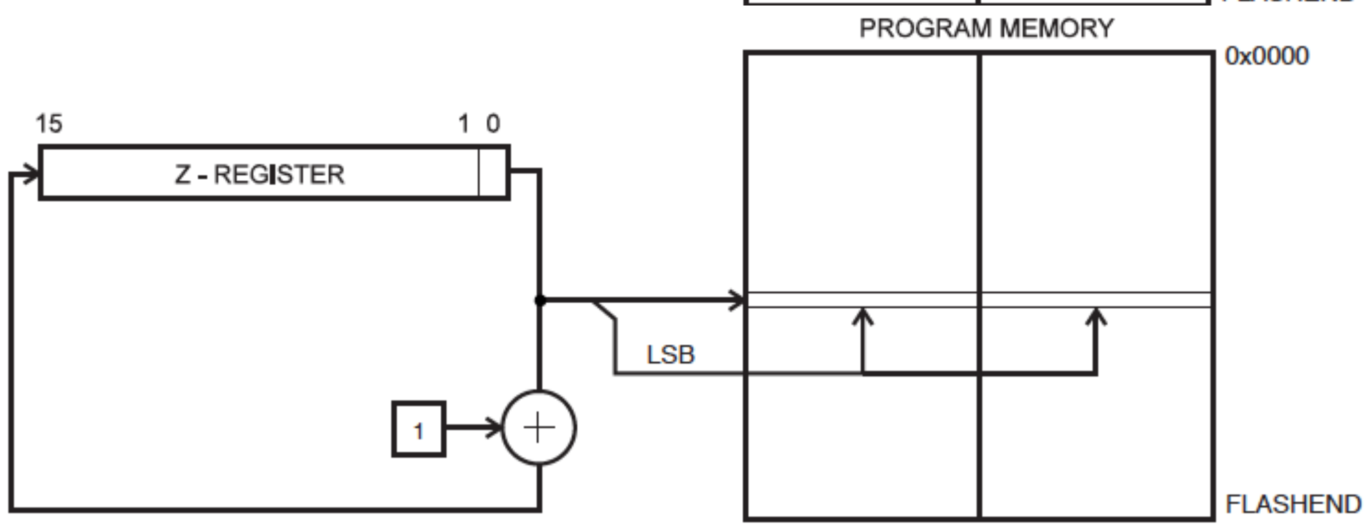
- Indirect data memory addressing with post-increment



Addressing Modes

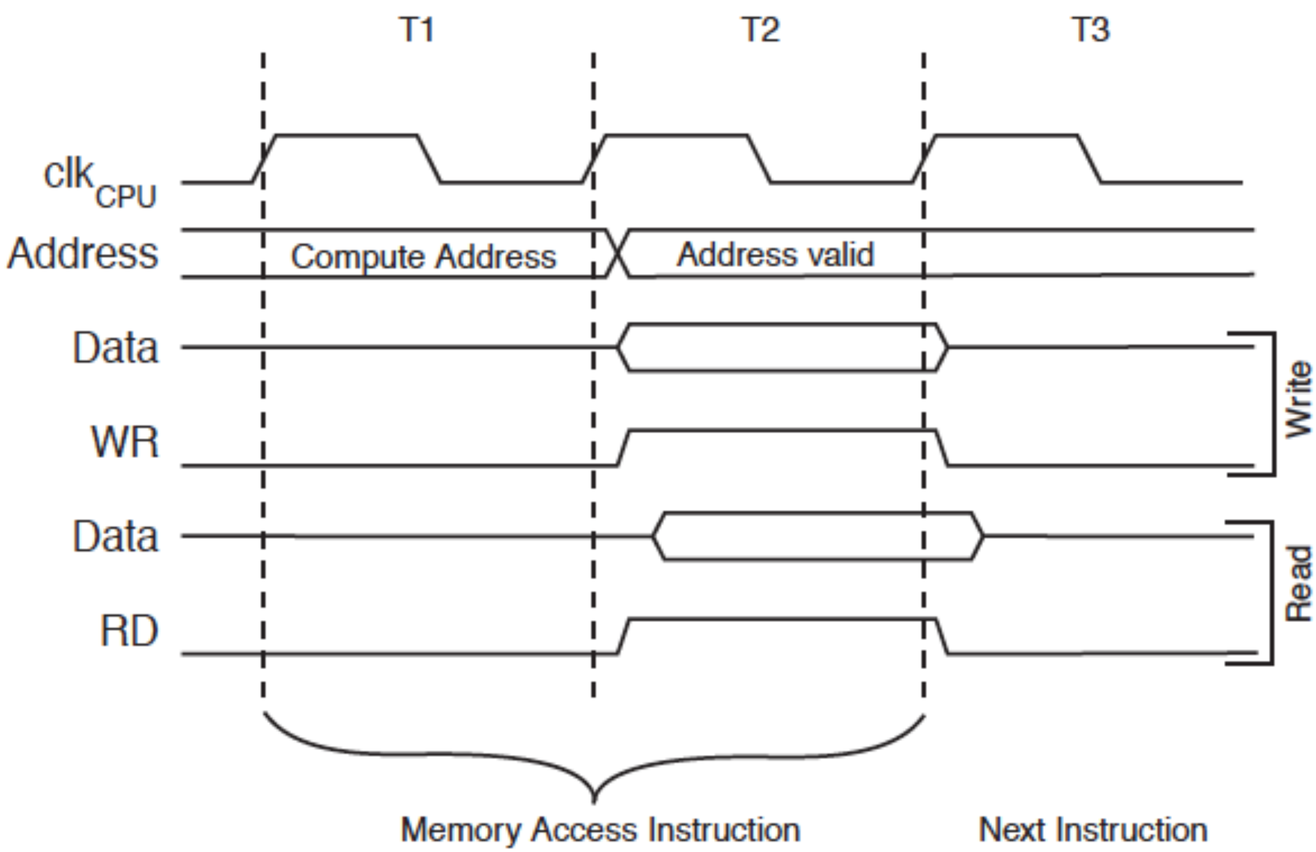
- Program memory addressing (constant data)





SRAM Read/Write Timing

Figure 7-4. On-chip Data SRAM Access Cycles



Stack Pointer Register

- Special register in I/O space [3E, 3D]
  - Enough bits to address data space
  - Initialized to RAMEND (address of highest memory address)

- Instructions that use the stack pointer

Instruction	Stack pointer	Description
PUSH	Decremented by 1	Data is pushed onto the stack
CALL ICALL RCALL	Decremented by 2	Return address is pushed onto the stack with a subroutine call or interrupt
POP	Incremented by 1	Data is popped from the stack
RET RETI	Incremented by 2	Return address is popped from the stack with return from subroutine or return from interrupt

Program Status Register (PSR)

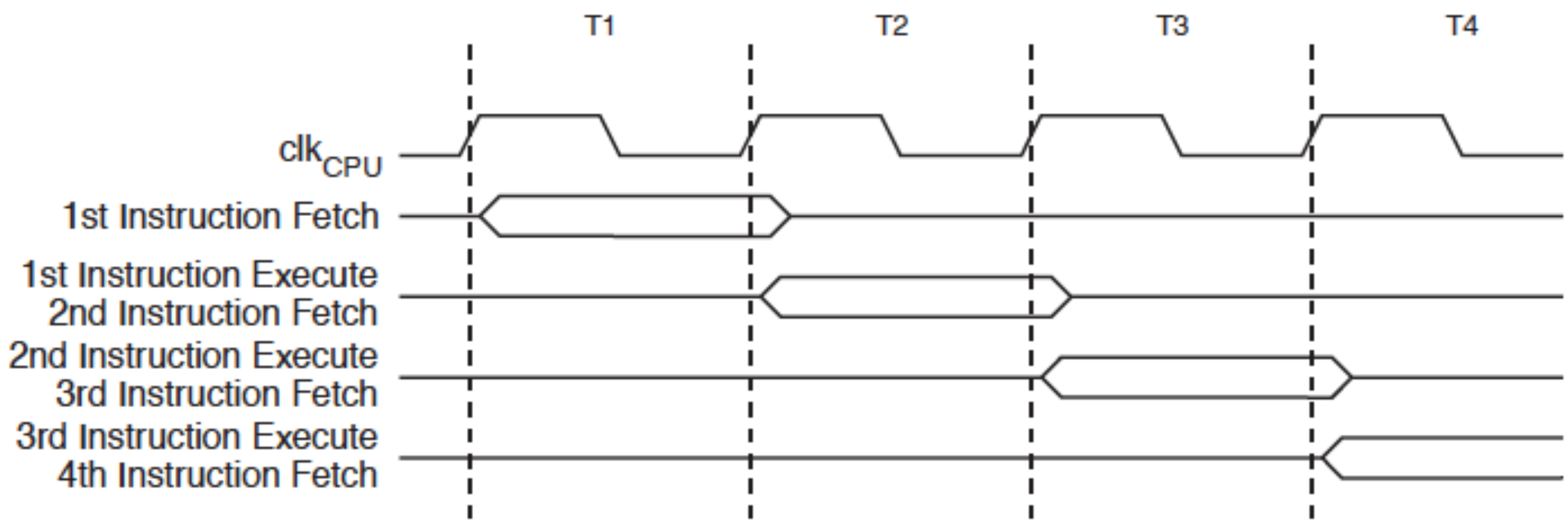
Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Status bits set by instructions/Checked by Branch/Skip instructions
  - I – Global interrupt enable
  - T – Flag bit
  - H – Half carry (BCD arithmetic)
  - S – Sign
  - V – Overflow
  - N – Negative
  - Z – Zero
  - C – Carry

Simple 2-Stage Pipeline

- Branch/Skip??

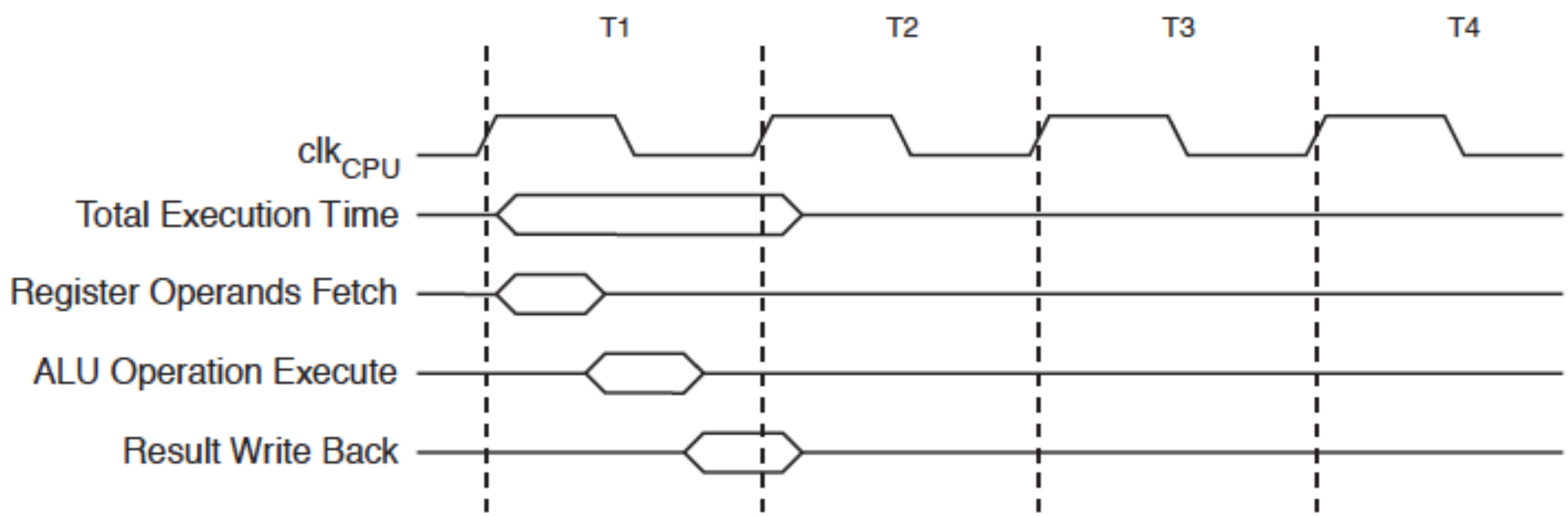
Figure 6-4. The Parallel Instruction Fetches and Instruction Executions



Single-Cycle ALU Instructions

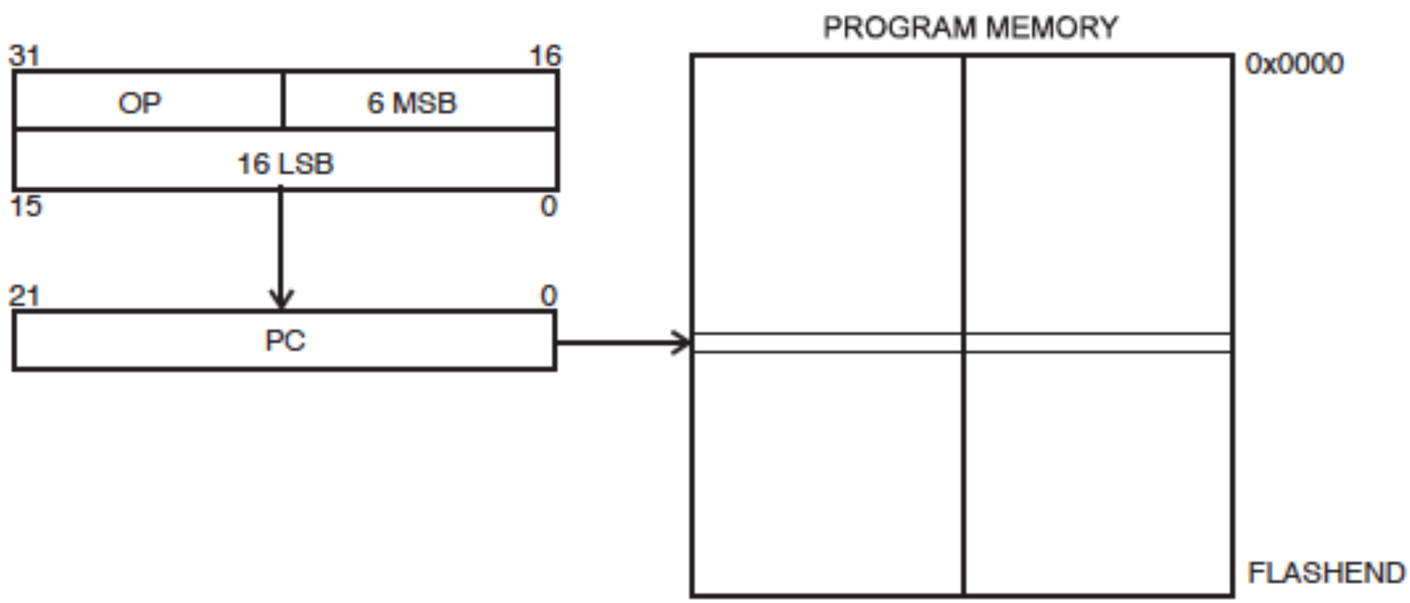
- Most instructions execute in one cycle
- Makes program timing calculations (relatively) easy
  - No cache misses
  - 1 clock/instruction

Figure 6-5. Single Cycle ALU Operation



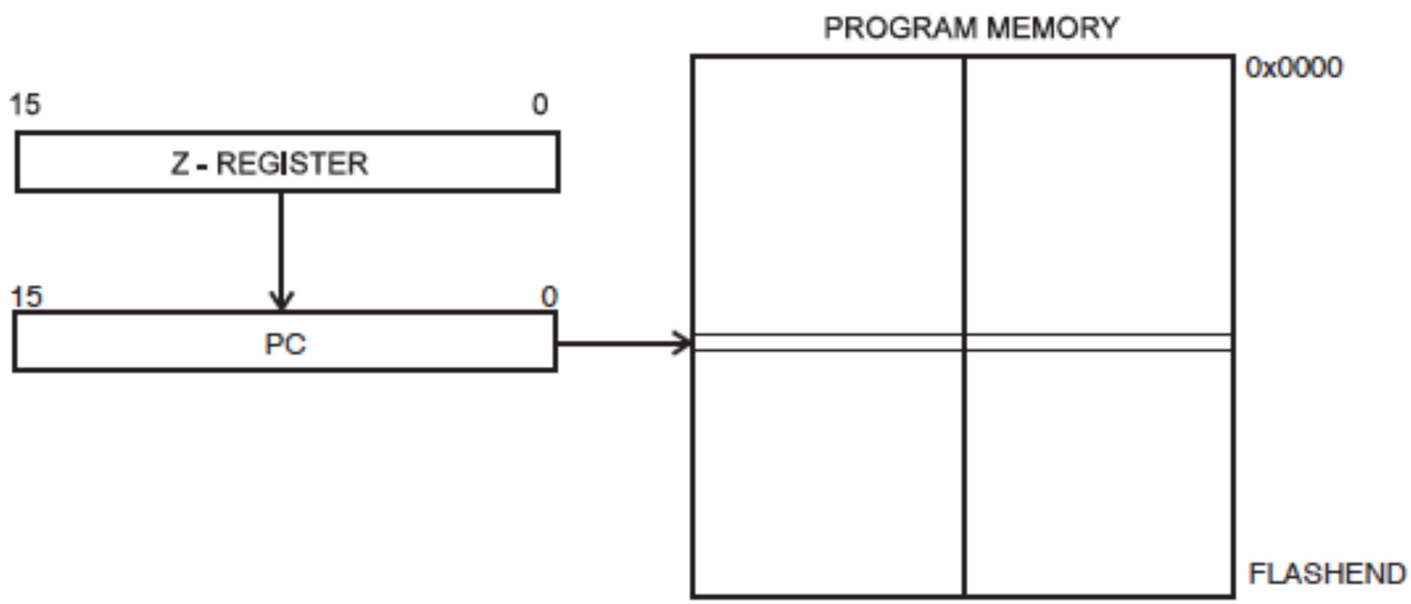
Addressing Modes

- JMP, CALL – Direct Program Memory Addressing



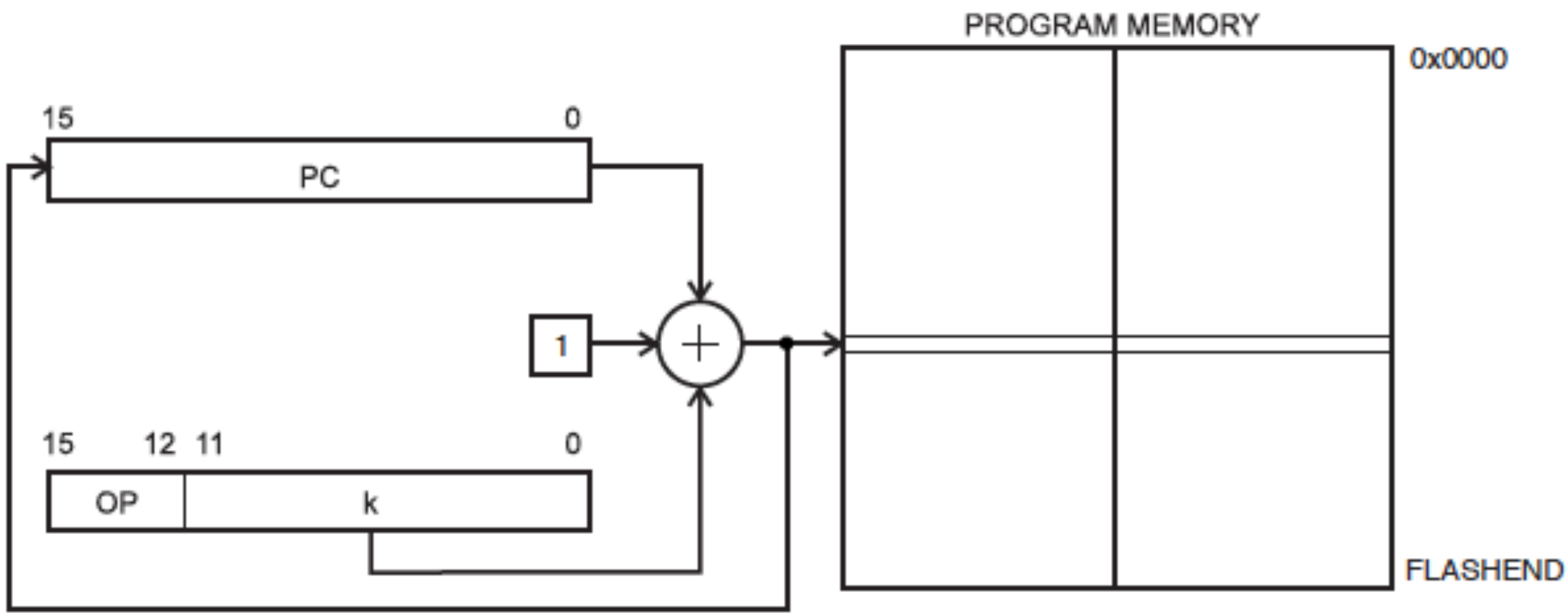
Addressing Modes

- IJMP, ICALL – Indirect program memory addressing



Addressing Modes

- RJMP, RCALL – Relative program memory addressing



Arithmetic Instructions

ADD	Rd, Rr	Add without Carry	Rd	←	Rd + Rr	Z,C,N,V,S,H	1
ADC	Rd, Rr	Add with Carry	Rd	←	Rd + Rr + C	Z,C,N,V,S,H	1
ADIW[1]	Rd, K	Add Immediate to Word	Rd	←	Rd + 1:Rd + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract without Carry	Rd	←	Rd - Rr	Z,C,N,V,S,H	1
SUBI	Rd, K	Subtract Immediate	Rd	←	Rd - K	Z,C,N,V,S,H	1
SBC	Rd, Rr	Subtract with Carry	Rd	←	Rd - Rr - C	Z,C,N,V,S,H	1
SBCI	Rd, K	Subtract Immediate with Carry	Rd	←	Rd - K - C	Z,C,N,V,S,H	1
SBIW[1]	Rd, K	Subtract Immediate from Word	Rd + 1:Rd	←	Rd + 1:Rd - K	Z,C,N,V,S	2
INC	Rd	Increment	Rd	←	Rd + 1	Z,N,V,S	1
DEC	Rd	Decrement	Rd	←	Rd - 1	Z,N,V,S	1
MUL[1]	Rd,Rr	Multiply Unsigned	R1:R0	←	Rd x Rr (UU)	Z,C	2
MULS[1]	Rd,Rr	Multiply Signed	R1:R0	←	Rd x Rr (SS)	Z,C	2
MULSU[1]	Rd,Rr	Multiply Signed with Unsigned	R1:R0	←	Rd x Rr (SU)	Z,C	2
FMUL[1]	Rd,Rr	Fractional Multiply Unsigned	R1:R0	←	Rd x Rr<<1 (UU)	Z,C	2
FMULS[1]	Rd,Rr	Fractional Multiply Signed	R1:R0	←	Rd x Rr<<1 (SS)	Z,C	2
FMULSU[1]	Rd,Rr	Fractional Multiply Signed with Unsigned	R1:R0	←	Rd x Rr<<1 (SU)	Z,C	2

Logical Instructions

AND	Rd, Rr	Logical AND	Rd	←	Rd • Rr	Z,N,V,S	1
ANDI	Rd, K	Logical AND with Immediate	Rd	←	Rd • K	Z,N,V,S	1
OR	Rd, Rr	Logical OR	Rd	←	Rd v Rr	Z,N,V,S	1
ORI	Rd, K	Logical OR with Immediate	Rd	←	Rd v K	Z,N,V,S	1
EOR	Rd, Rr	Exclusive OR	Rd	←	Rd ⊕ Rr	Z,N,V,S	1
COM	Rd	One's Complement	Rd	←	$FF - Rd	Z,C,N,V,S	1
NEG	Rd	Two's Complement	Rd	←	$00 - Rd	Z,C,N,V,S,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd	←	Rd v K	Z,N,V,S	1
CBR	Rd,K	Clear Bit(s) in Register	Rd	←	Rd • ($FFh - K)	Z,N,V,S	1
TST	Rd	Test for Zero or Minus	Rd	←	Rd • Rd	Z,N,V,S	1
CLR	Rd	Clear Register	Rd	←	Rd ⊕ Rd	Z,N,V,S	1
SER	Rd	Set Register	Rd	←	$FF	None	1

Jump and Call Instructions

RJMP	k	Relative Jump	PC	←	PC + k + 1	None	2
IJMP[1]		Indirect Jump to (Z)	PC(15:0) PC(21:16)	← ←	Z, 0	None	2
EIJMP[1]		Extended Indirect Jump to (Z)	PC(15:0) PC(21:16)	← ←	Z, EIND	None	2
JMP[1]	k	Jump	PC	←	k	None	3
RCALL	k	Relative Call Subroutine	PC	←	PC + k + 1	None	3 / 4[3][5]
ICALL[1]		Indirect Call to (Z)	PC(15:0) PC(21:16)	← ←	Z, 0	None	3 / 4[3]
EICALL[1]		Extended Indirect Call to (Z)	PC(15:0) PC(21:16)	← ←	Z, EIND	None	4 [3]
CALL[1]	k	call Subroutine	PC	←	k	None	4 / 5[3]
RET		Subroutine Return	PC	←	STACK	None	4 / 5[3]
RETI		Interrupt Return	PC	←	STACK	I	4 / 5[3]

Skip and Branch Instructions

CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC	←	PC + 2 or 3	None	1 / 2 / 3
CP	Rd,Rr	Compare	Rd - Rr			Z,C,N,V,S,H	1
CPC	Rd,Rr	Compare with Carry	Rd - Rr - C			Z,C,N,V,S,H	1
CPI	Rd,K	Compare with Immediate	Rd - K			Z,C,N,V,S,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b) = 0) PC	←	PC + 2 or 3	None	1 / 2 / 3
SBRS	Rr, b	Skip if Bit in Register Set	if (Rr(b) = 1) PC	←	PC + 2 or 3	None	1 / 2 / 3
SBIC	A, b	Skip if Bit in I/O Register Cleared	if (I/O(A,b) = 0) PC	←	PC + 2 or 3	None	1 / 2 / 3
SBIS	A, b	Skip if Bit in I/O Register Set	If (I/O(A,b) =1) PC	←	PC + 2 or 3	None	1 / 2 / 3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC	←	PC + k + 1	None	1 / 2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC	←	PC + k + 1	None	1 / 2
BREQ	k	Branch if Equal	if (Z = 1) then PC	←	PC + k + 1	None	1 / 2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC	←	PC + k + 1	None	1 / 2
BRCS	k	Branch if Carry Set	if (C = 1) then PC	←	PC + k + 1	None	1 / 2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC	←	PC + k + 1	None	1 / 2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC	←	PC + k + 1	None	1 / 2
BRLO	k	Branch if Lower	if (C = 1) then PC	←	PC + k + 1	None	1 / 2

Skip and Branch (cont)

BRMI	k	Branch if Minus	if (N = 1) then PC	←	PC + k + 1	None	1 / 2
BRPL	k	Branch if Plus	if (N = 0) then PC	←	PC + k + 1	None	1 / 2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC	←	PC + k + 1	None	1 / 2
BRLT	k	Branch if Less Than, Signed	if (N ⊕ V= 1) then PC	←	PC + k + 1	None	1 / 2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC	←	PC + k + 1	None	1 / 2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC	←	PC + k + 1	None	1 / 2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC	←	PC + k + 1	None	1 / 2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC	←	PC + k + 1	None	1 / 2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC	←	PC + k + 1	None	1 / 2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC	←	PC + k + 1	None	1 / 2
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC	←	PC + k + 1	None	1 / 2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC	←	PC + k + 1	None	1 / 2

Move, Load

LDI	Rd, K	Load Immediate	Rd ← K	None	1
LDS[1]	Rd, k	Load Direct from data space	Rd ← (k)	None	1[5]/2[3]
LD[2]	Rd, X	Load Indirect	Rd ← (X)	None	1[5]2[3]
LD[2]	Rd, X+	Load Indirect and Post-Increment	Rd ← (X) X ← X + 1	None	2[3]
LD[2]	Rd, -X	Load Indirect and Pre-Decrement	X ← X - 1, ← X - 1 Rd ← (X) ← (X)	None	2[3]/3[5]
LD[2]	Rd, Y	Load Indirect	Rd ← (Y) ← (Y)	None	1[5]/2[3]
LD[2]	Rd, Y+	Load Indirect and Post-Increment	Rd ← (Y) Y ← Y + 1	None	2[3]
LD[2]	Rd, -Y	Load Indirect and Pre-Decrement	Y ← Y - 1 Rd ← (Y)	None	2[3]/3[5]
LDD[1]	Rd, Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None	2[3]
LD[2]	Rd, Z	Load Indirect	Rd ← (Z)	None	1[5]/2[3]
LD[2]	Rd, Z+	Load Indirect and Post-Increment	Rd ← (Z), Z ← Z+1	None	2[3]
LD[2]	Rd, -Z	Load Indirect and Pre-Decrement	Z ← Z - 1, Rd ← (Z)	None	2[3]/3[5]
LDD[1]	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None	2[3]

Store

STS[1]	k, Rr	Store Direct to Data Space	(k)	←	Rd	None	$1^{(5)}/2^{(3)}$
ST[2]	X, Rr	Store Indirect	(X)	←	Rr	None	$1^{(5)}/2^{(3)}$
ST[2]	X+, Rr	Store Indirect and Post-Increment	(X) X	← ←	Rr, X + 1	None	$1^{(5)}/2^{(3)}$
ST[2]	-X, Rr	Store Indirect and Pre-Decrement	X (X)	← ←	X - 1, Rr	None	$2^{(3)}$
ST[2]	Y, Rr	Store Indirect	(Y)	←	Rr	None	$1^{(5)}/2^{(3)}$
ST[2]	Y+, Rr	Store Indirect and Post-Increment	(Y) Y	← ←	Rr, Y + 1	None	$1^{(5)}/2^{(3)}$
ST[2]	-Y, Rr	Store Indirect and Pre-Decrement	Y (Y)	← ←	Y - 1, Rr	None	$2^{(3)}$
STD[1]	Y+q, Rr	Store Indirect with Displacement	(Y + q)	←	Rr	None	$2^{(3)}$
ST[2]	Z, Rr	Store Indirect	(Z)	←	Rr	None	$1^{(5)}/2^{(3)}$
ST[2]	Z+, Rr	Store Indirect and Post-Increment	(Z) Z	← ←	Rr Z + 1	None	$1^{(5)}/2^{(3)}$
ST[2]	-Z, Rr	Store Indirect and Pre-Decrement	Z	←	Z - 1	None	$2^{(3)}$
STD[1]	Z+q,Rr	Store Indirect with Displacement	(Z + q)	←	Rr	None	$2^{(3)}$

Load/Store Program Memory

LPM[1][2]		Load Program Memory	R0	←	(Z)	None	3
LPM[1][2]	Rd, Z	Load Program Memory	Rd	←	(Z)	None	3
LPM[1][2]	Rd, Z+	Load Program Memory and Post-Increment	Rd Z	← ←	(Z), Z + 1	None	3
ELPM[1]		Extended Load Program Memory	R0	←	(RAMPZ:Z)	None	3
ELPM[1]	Rd, Z	Extended Load Program Memory	Rd	←	(RAMPZ:Z)	None	3
ELPM[1]	Rd, Z+	Extended Load Program Memory and Post-Increment	Rd Z	← ←	(RAMPZ:Z), Z + 1	None	3
SPM[1]		Store Program Memory	(RAMPZ:Z)	←	R1:R0	None	-
SPM[1]	Z+	Store Program Memory and Post-Increment by 2	(RAMPZ:Z) Z	← ←	R1:R0, Z + 2	None	-

Move, I/O, Push/Pop

MOV	Rd, Rr	Copy Register	Rd	←	Rr	None	1
MOVW[1]	Rd, Rr	Copy Register Pair	Rd+1:Rd	←	Rr+1:Rr	None	1
IN	Rd, A	In From I/O Location	Rd	←	I/O(A)	None	1
OUT	A, Rr	Out To I/O Location	I/O(A)	←	Rr	None	1
PUSH[1]	Rr	Push Register on Stack	STACK	←	Rr	None	2
POP[1]	Rd	Pop Register from Stack	Rd	←	STACK	None	2

Shift and Bit Instructions

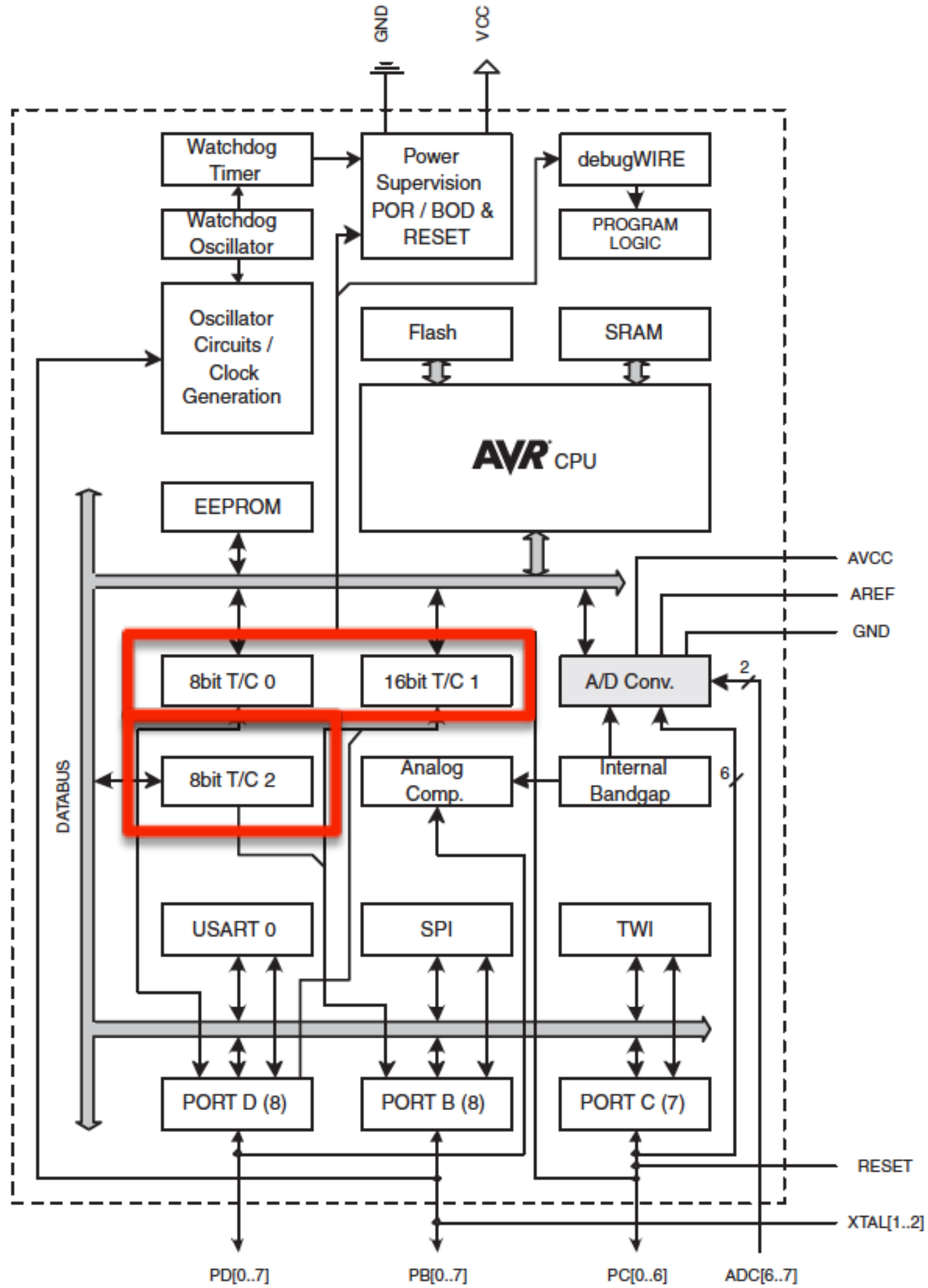
LSL	Rd	Logical Shift Left	Rd(n+1) Rd(0) C	← ← ←	Rd(n), 0, Rd(7)	Z,C,N,V,H	1
LSR	Rd	Logical Shift Right	Rd(n) Rd(7) C	← ← ←	Rd(n+1), 0, Rd(0)	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0) Rd(n+1) C	← ← ←	C, Rd(n), Rd(7)	Z,C,N,V,H	1
ROR	Rd	Rotate Right Through Carry	Rd(7) Rd(n) C	← ← ←	C, Rd(n+1), Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n)	←	Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0)	↔	Rd(7..4)	None	1
BSET	s	Flag Set	SREG(s)	←	1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s)	←	0	SREG(s)	1
SBI	A, b	Set Bit in I/O Register	I/O(A, b)	←	1	None	1[(5)]2
CBI	A, b	Clear Bit in I/O Register	I/O(A, b)	←	0	None	1[(5)]/2
BST	Rr, b	Bit Store from Register to T	T	←	Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b)	←	T	None	1
SEC		Set Carry	C	←	1	C	1
CLC		Clear Carry	C	←	0	C	1
SEN		Set Negative Flag	N	←	1	N	1
CLN		Clear Negative Flag	N	←	0	N	1
SEZ		Set Zero Flag	Z	←	1	Z	1
CLZ		Clear Zero Flag	Z	←	0	Z	1

Bit Instructions (cont)

SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Two's Complement Overflow	V ← 1	V	1
CLV		Clear Two's Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ← 1	H	1
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1
MCU Control Instructions					
BREAK[1]		Break	(See specific descr. for BREAK)	None	1
NOP		No Operation		None	1
SLEEP		Sleep	(see specific descr. for Sleep)	None	1
WDR		Watchdog Reset	(see specific descr. for WDR)	None	1

AVR Architecture

- Three timers
- Very flexible
  - Choose clock rate
  - Choose "roll-over" value
  - Generate interrupts
  - Generate PWM signals
    - (represent 8-bit value with using a clock signal)
- More in next lecture…

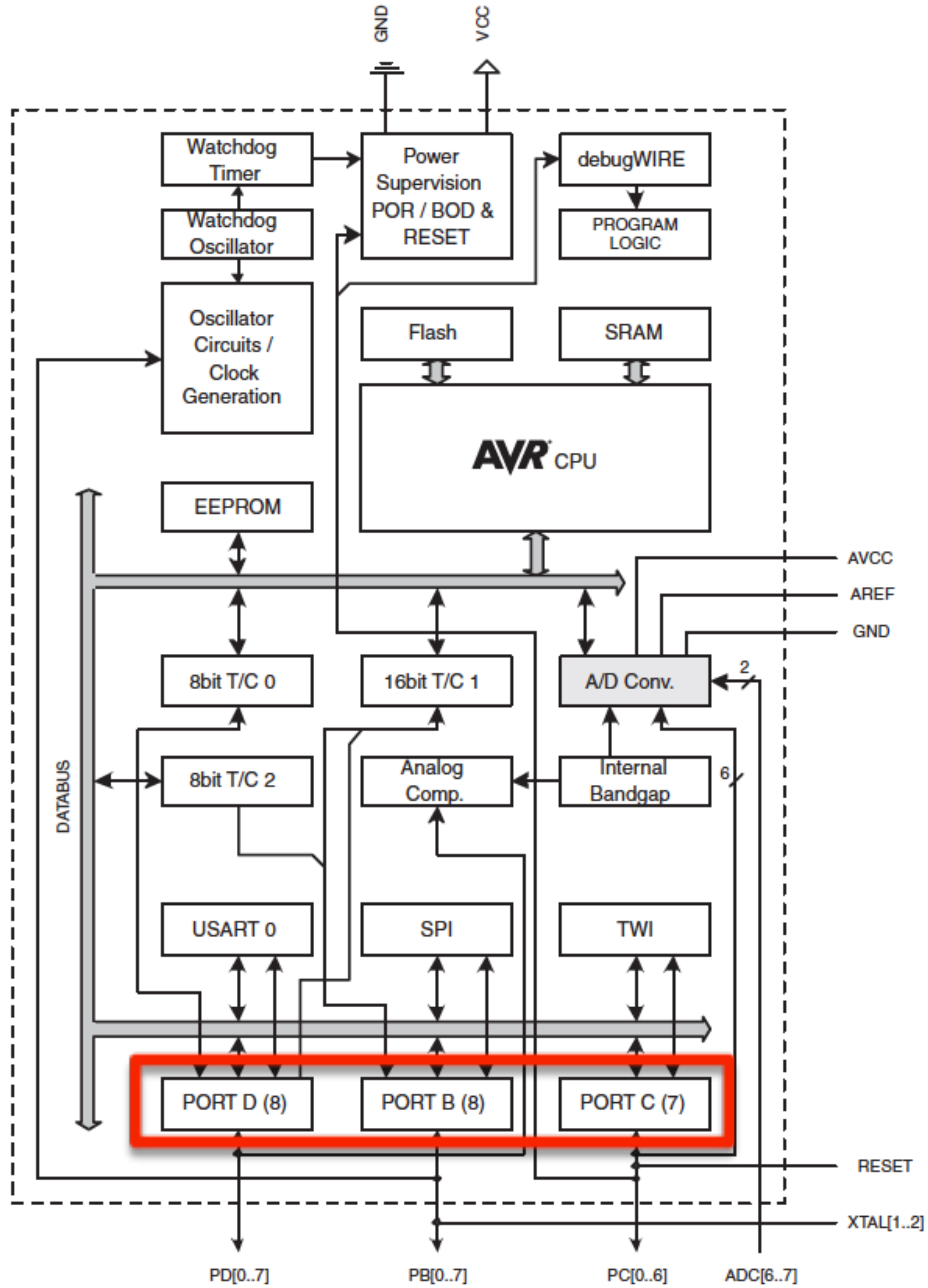


Arduino Timing Functions

- delay(*ms*)
  - wait for ms milliseconds before continuing
- delayMicroseconds(*us*)
  - wait for us microseconds before continuing
- unsigned long millis( )
  - return number of milliseconds since program started
- unsigned long micros( )
  - return number of microseconds since program started
  - resolution of 4 microseconds

AVR Architecture

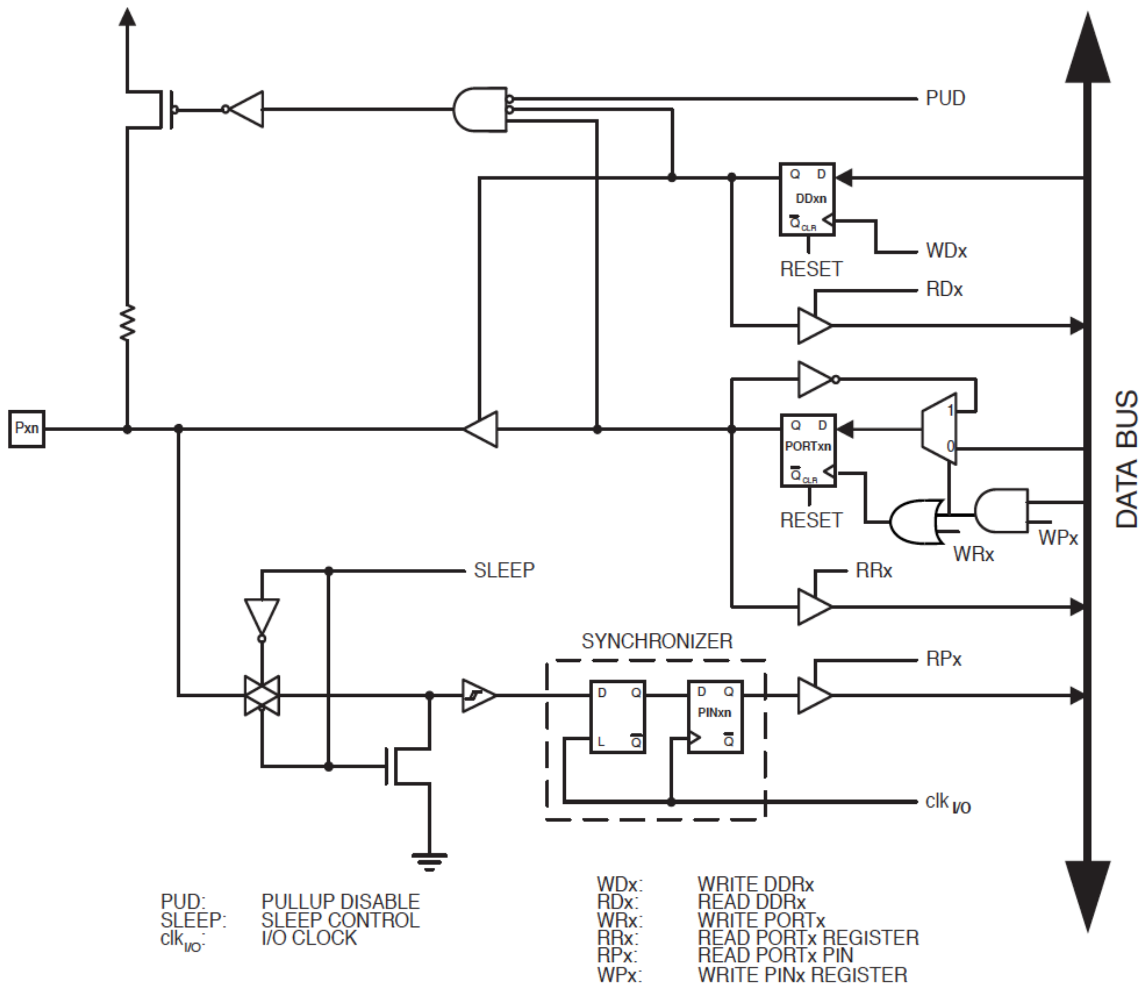
- Interface to pins
- Each pin directly programmable
  - Program direction
  - Program value
  - Program pull-ups
- Some pins are special
  - Analog vs. Digital
  - Clocks
  - Reset



I/O Ports

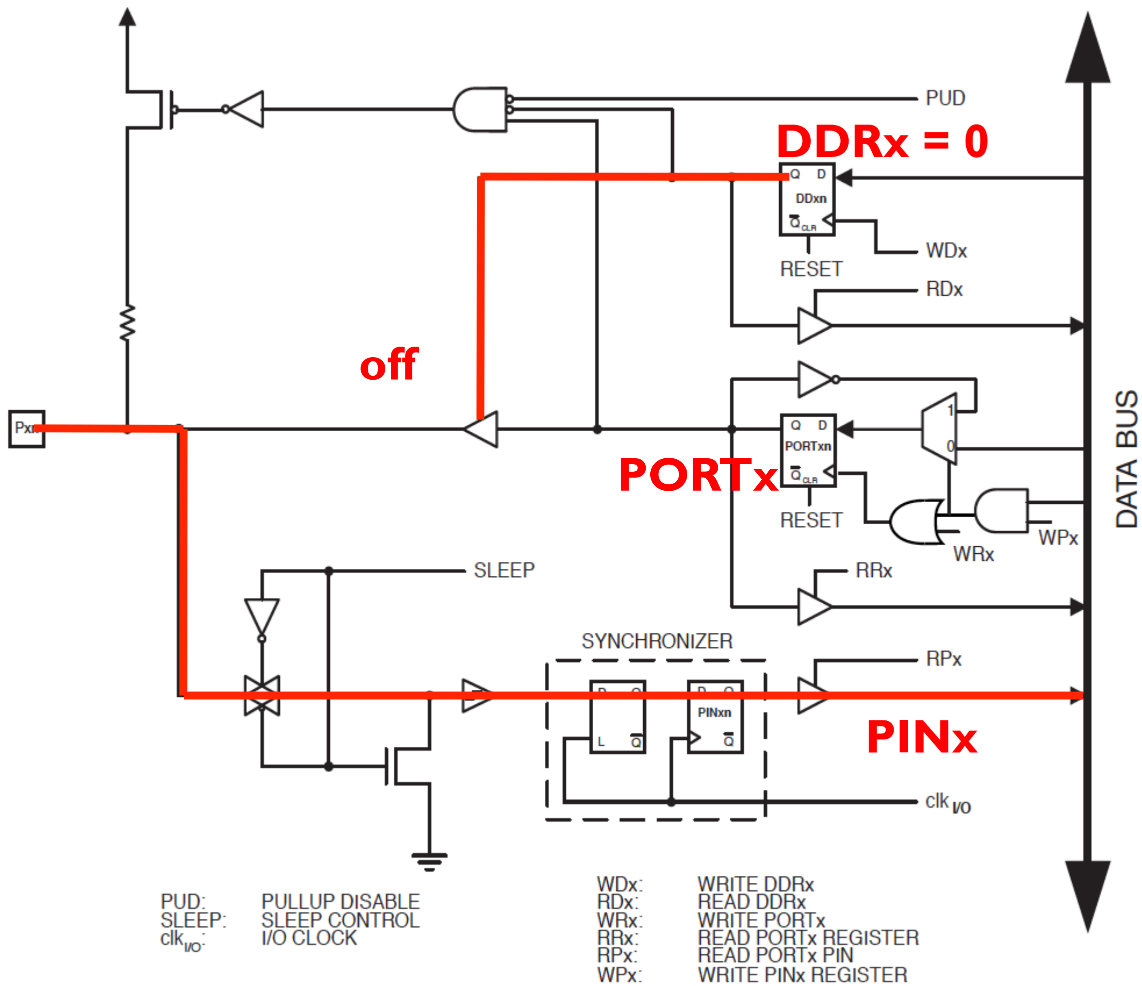
- 3 8-bit Ports (B, C, D)
- Each port controlled by 3 8-bit registers
  - Each bit controls one I/O pin
  - DDRx – Direction register
    - Defines whether a pin is an input (0) or and output (1)
  - PINx – Pin input value
    - Reading this "register" returns value of pin
  - PORTx – Pin output value
    - Writing this register sets value of pin

Pin Circuitry



Note: 1. WRx, WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $clk_{I/O}$, SLEEP, and PUD are common to all ports.

Pin Input

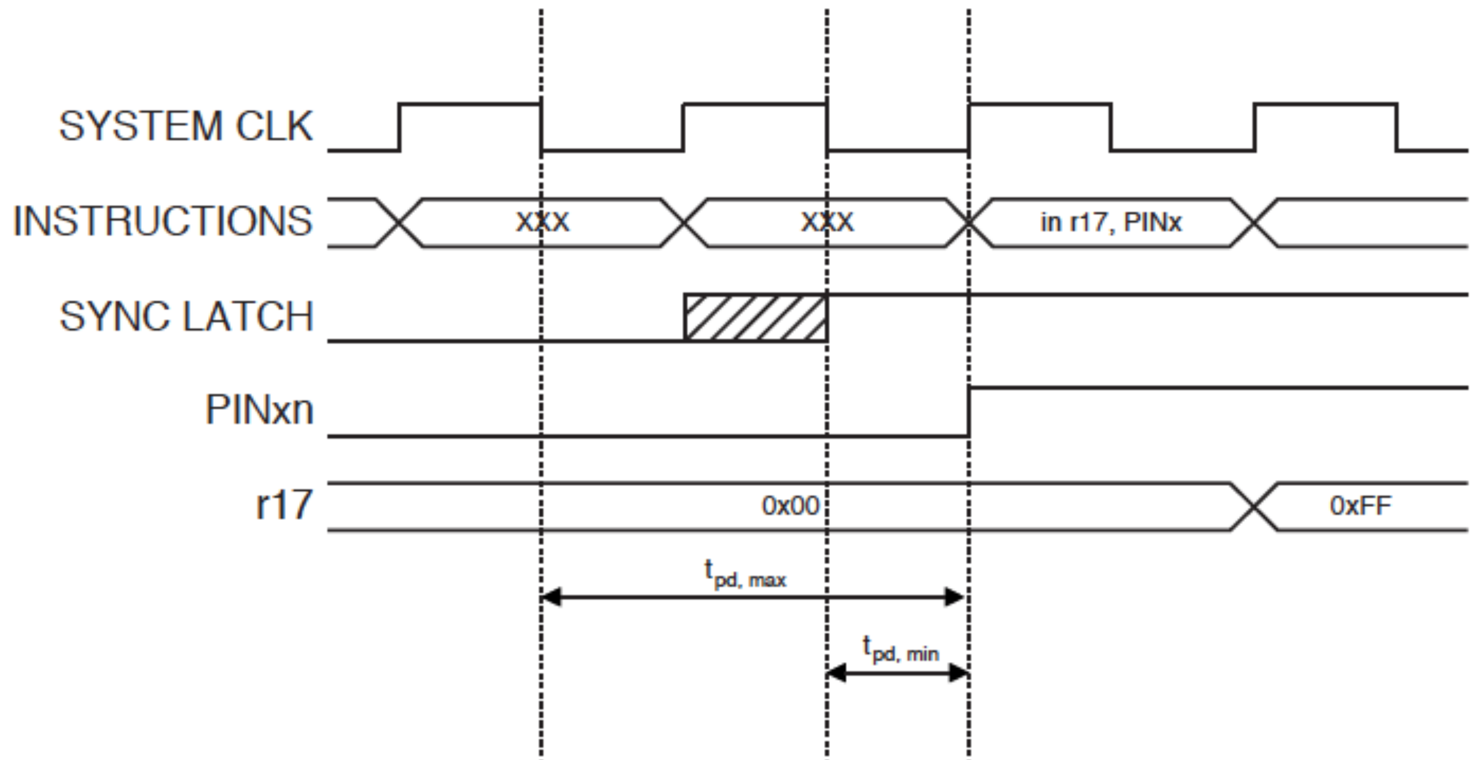


Note: 1. WRx, WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $clk_{I/O}$, SLEEP, and PUD are common to all ports.

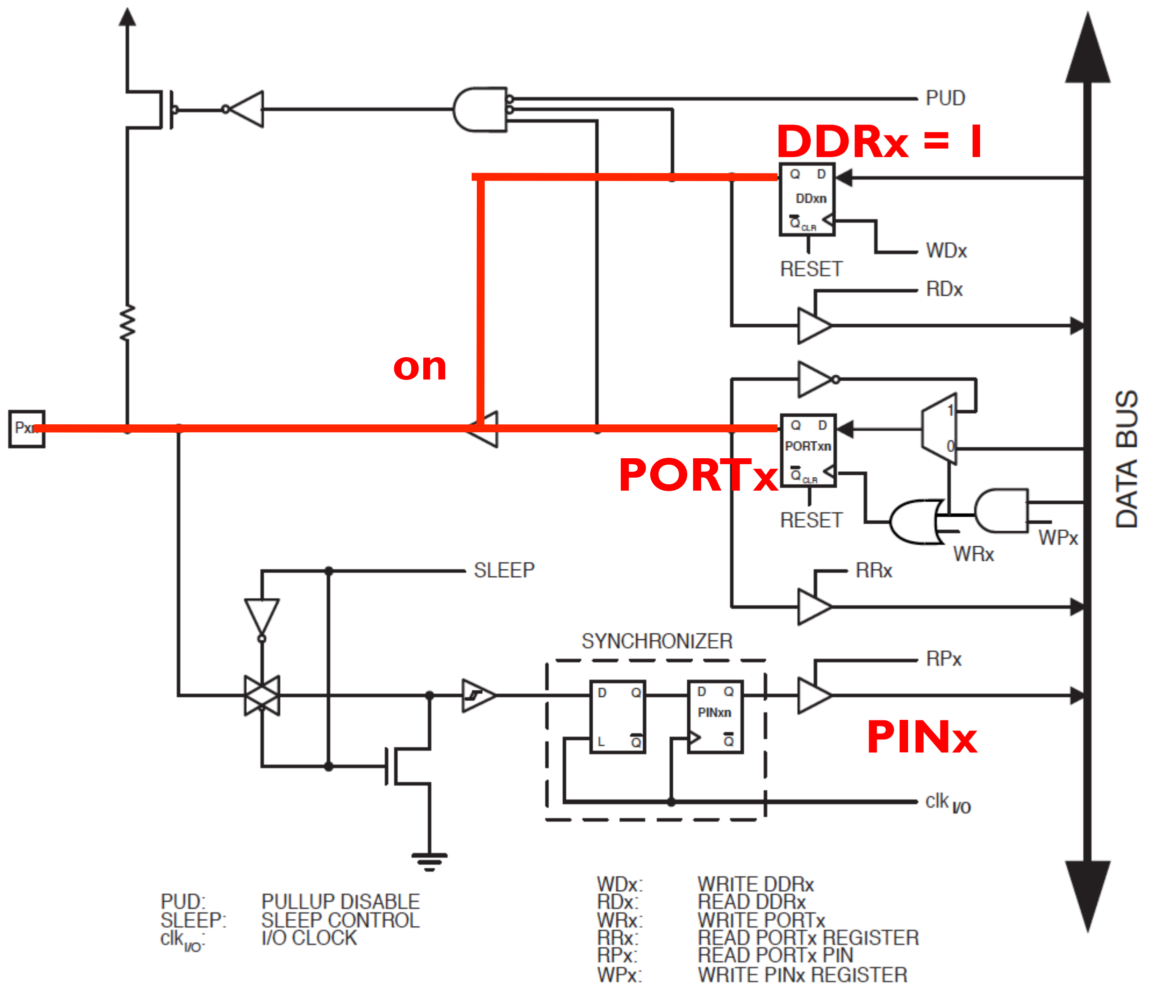
Synchronization Timing

- Note: Takes a clock cycle for data output to be reflected on the input

Figure 13-3. Synchronization when Reading an Externally Applied Pin value

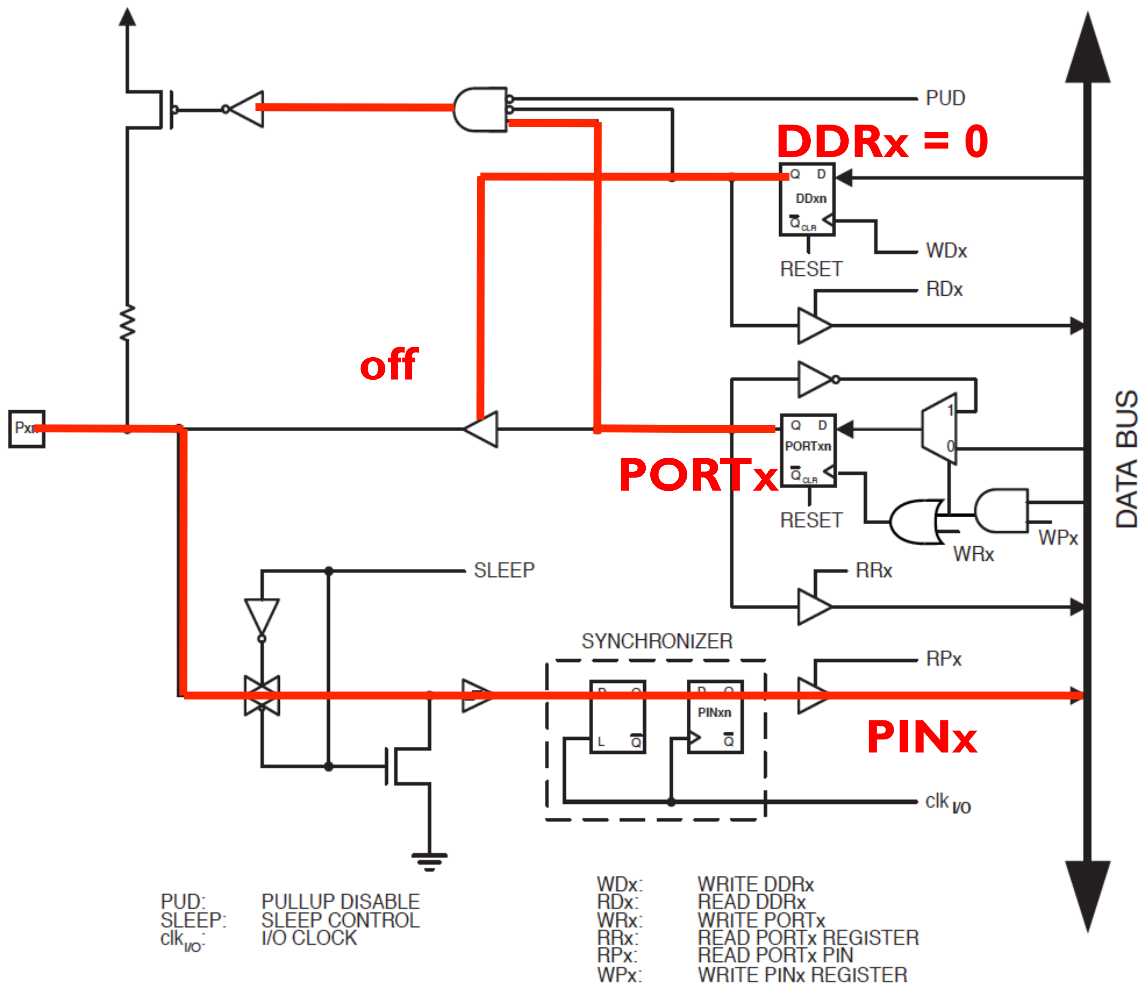


Pin Output



Note: 1. WRx, WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $clk_{I/O}$, SLEEP, and PUD are common to all ports.

Pin Input – PORT controls pullup

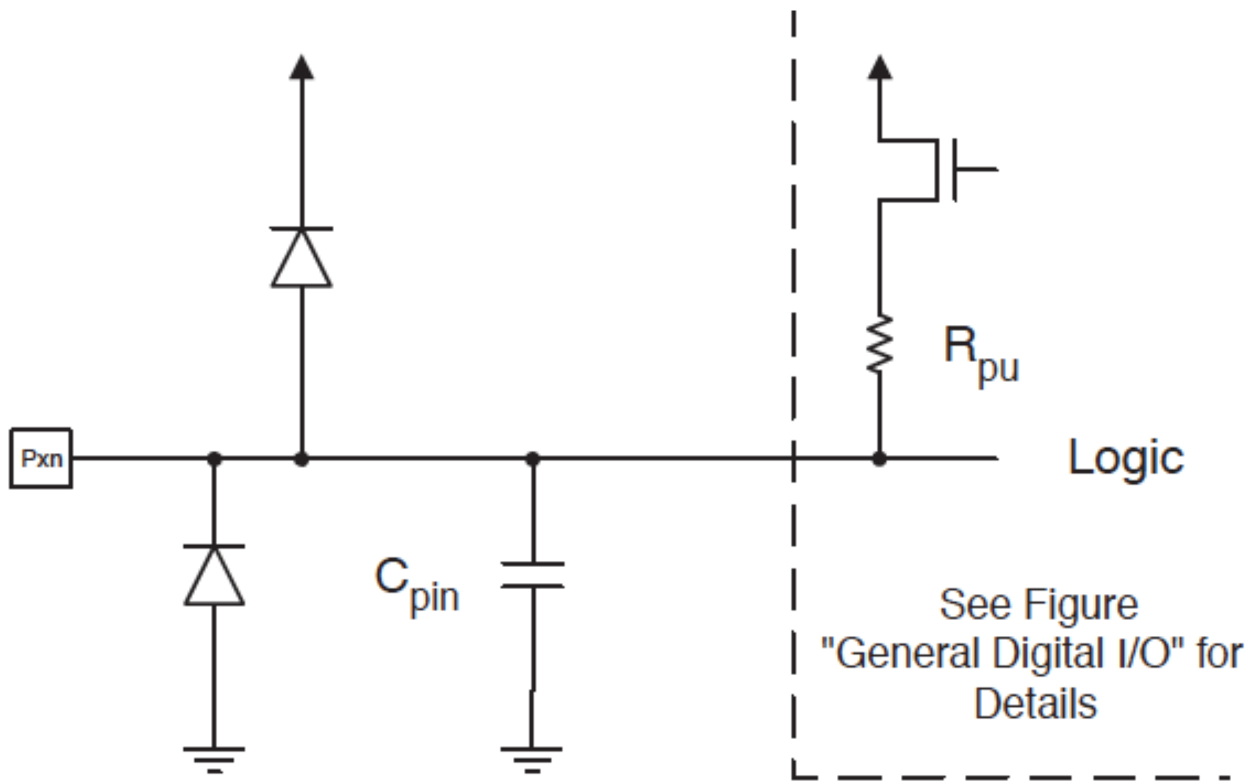


Note: 1. WRx, WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $clk_{I/O}$, SLEEP, and PUD are common to all ports.

I/O Ports

- Pullups
  - If a pin is an input (DDRxi = 0):
    - PORTxi = 0 – pin is floating
    - PORTxi = 1 – connects a pullup to the pin
      - Keeps pin from floating if noone driving
      - Allows wired-OR bus
- Individual bits can be set cleared using bit-ops
- A bit can be toggled by writing 1 to PINxi
  - SBI instruction e.g.

I/O Protection



Arduino Digital and Analog I/O Pins

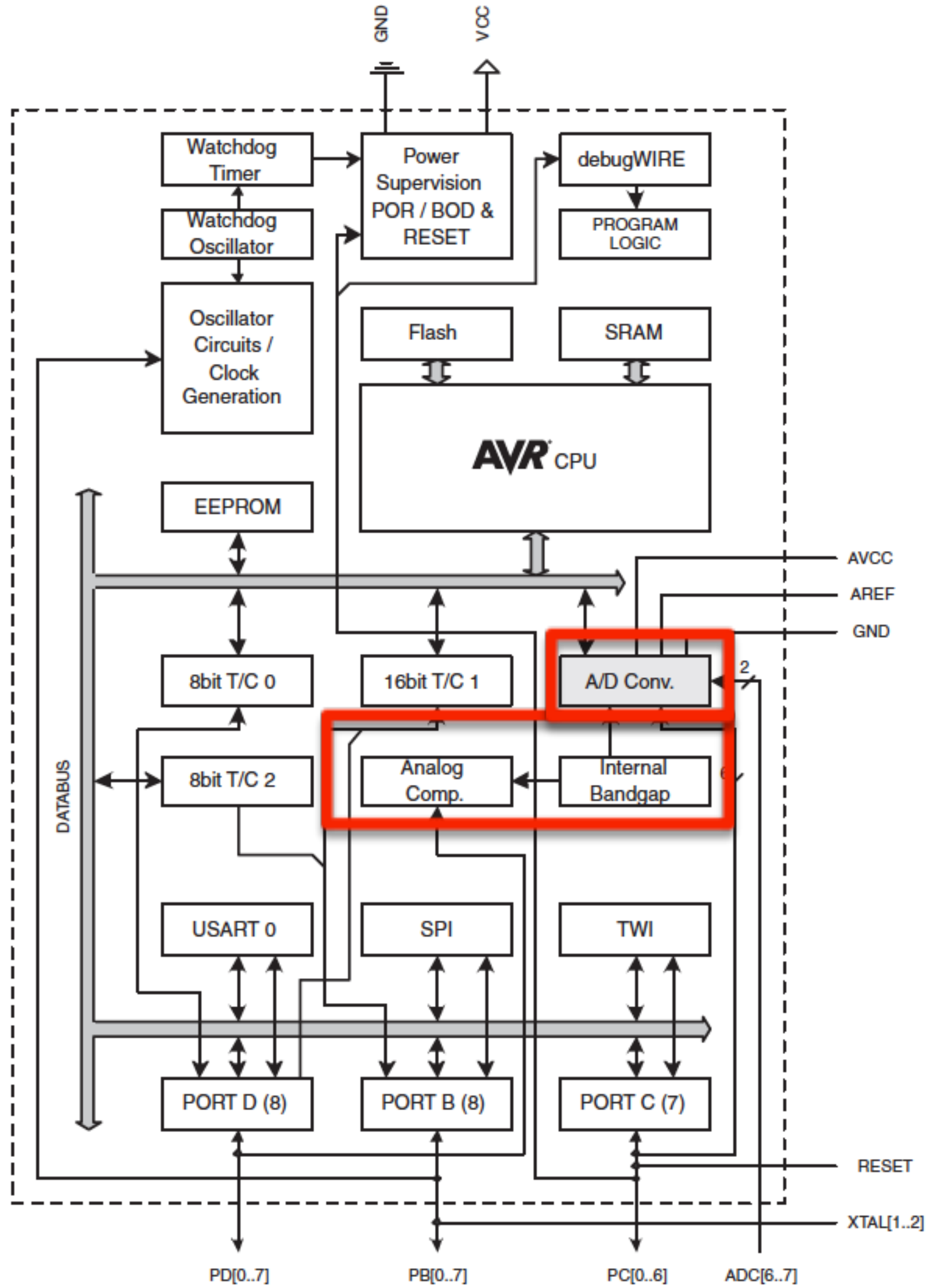
- Digital pins:
  - Pins 0 – 7: PORT D [0:7]
  - Pins 8 – 13: PORT B [0:5]
  - Pins 14 – 19: PORT C [0:5] (Arduino analog pins 0 – 5)
  - digital pins 0 and 1 are RX and TX for serial communication
  - digital pin 13 connected to the base board LED
- Digital Pin I/O Functions
  - pinMode(*pin*, *mode*)
    - Sets pin to INPUT or OUTPUT mode
    - Writes 1 bit in the DDRx register
  - digitalWrite(pin, value)
    - Sets pin value to LOW or HIGH (0 or 1)
    - Writes 1 bit in the PORTx register
  - int value = digitalRead(pin)
    - Reads back pin value (0 or 1)
    - Read 1 bit in the PINx register

Arduino Analog I/O

- Analog input pins: 0 – 5
- Analog output pins: 3, 5, 6, 9, 10, 11 (digital pins)
- Analog input functions
  - int val = analogRead(*pin*)
  - Converts 0 – 5v. voltage to a 10-bit number (0 – 1023)
  - Don't use pinMode
  - analogReference(*type*)
    - Used to change how voltage is converted (advanced)
- Analog output
  - analogWrite(*pin*, *value*)
    - value is 0 – 255
  - Generates a PWM output on digital pin (3, 5, 6, 9, 10, 11)
  - @490Hz frequency

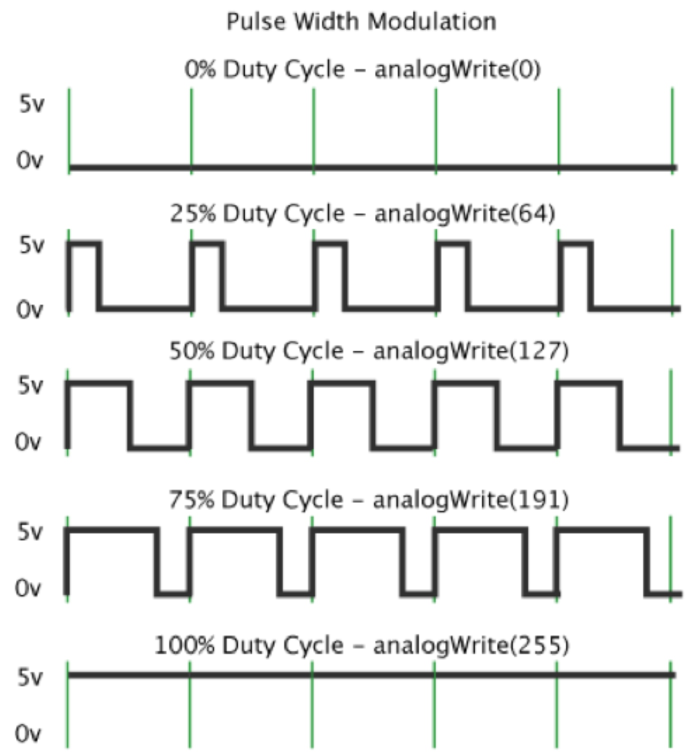
AVR Architecture

- Analog inputs
- Convert voltage to a 10-bit digital value
- Can provide reference voltages



PWM – Pulse Width Modulation

- Use one wire to represent a multi-bit value
  - A clock with a variable duty cycle
  - Duty cycle used to represent value
  - We can turn it into a analog voltage using an integrating filter



Port Special Functions

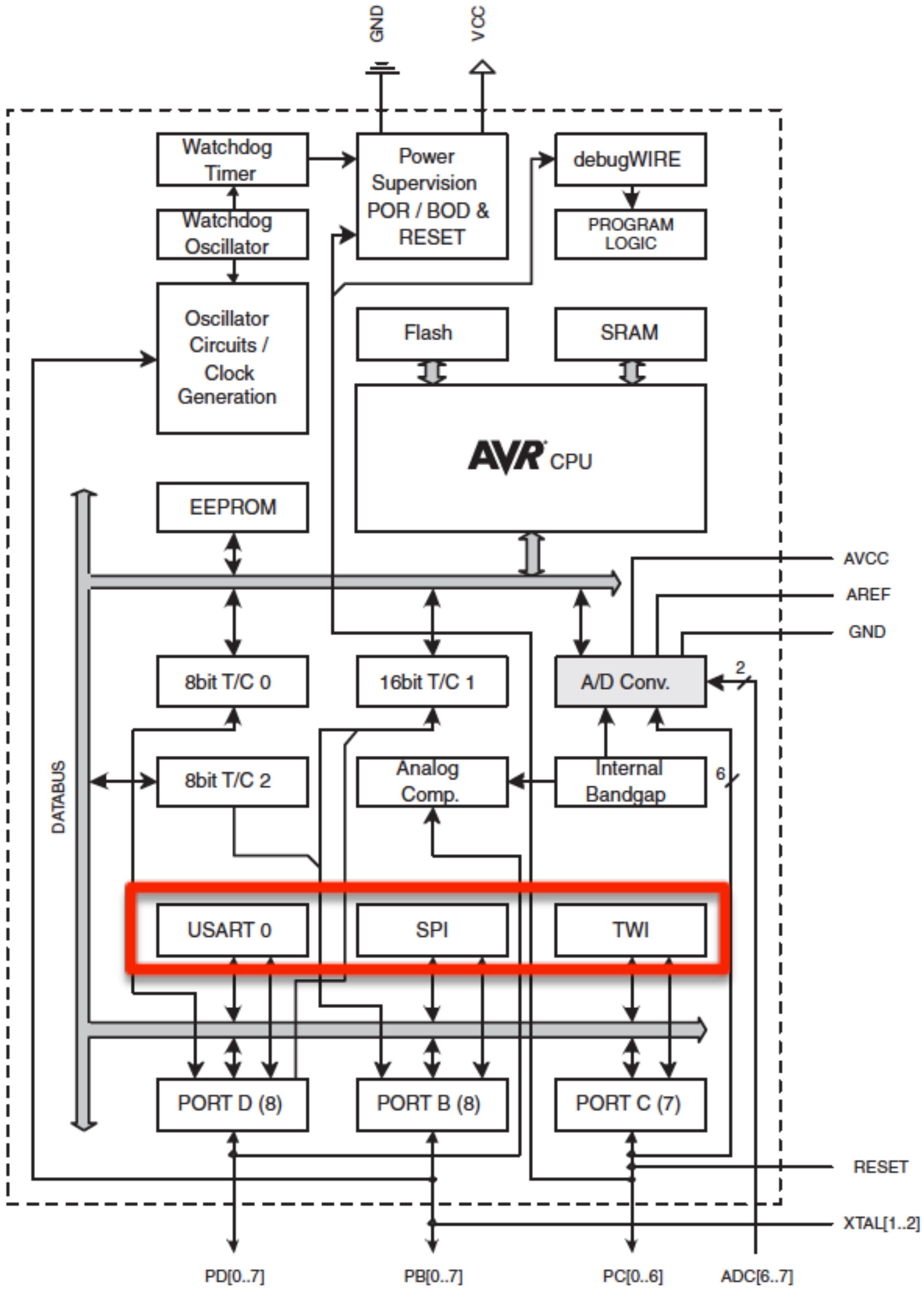
- Lots of special uses for pins
  - Clock connections
  - Timer connections
    - e.g. comparator output for PWM
  - Interrupts
  - Analog references
  - Serial bus I/Os
    - USART
    - PCI

Reading and Writing Pins Directly

- Only one pin can be changed using the Arduino I/O functions
  - Setting multiple pins takes time and instructions
- To change multiple pins simultaneously, directly read/write the pin registers
  - DDR{A/B/C}
  - PORT{A/B/C}
  - PIN{A/B/C}
- e.g. to set all digital pins 0 – 7 to a value:
  - PORTD = B01100101;

AVR Architecture

- Special I/O support
  - Serial protocols
- Uses special pins
- Uses timers
- Beyond scope of this course



Arduino C Programs

- Arduino calls these "sketches"
  - Basically C with libraries
- Program structure
  - Header: declarations, includes, etc.
  - setup()
  - loop()
- Setup is like Verilog initial
  - executes once when program starts
- loop() is like Verilog always
  - continuously re-executed when the end is reached

Blink Program

```
int ledPin =  13;    // LED connected to digital pin 13

// The setup() method runs once, when the sketch starts

void setup()   {
  // initialize the digital pin as an output:
  pinMode(ledPin, OUTPUT);
}

// the loop() method runs over and over again,
// as long as the Arduino has power

void loop()
{
  digitalWrite(ledPin, HIGH);   // set the LED on
  delay(1000);                  // wait for a second
  digitalWrite(ledPin, LOW);    // set the LED off
  delay(1000);                  // wait for a second
}
```

The Arduino C++ Main Program

```
int main(void)
{
    init();

    setup();

    for (;;)
        loop();

    return 0;
}
```

Arduino Serial I/O

- Communication with PC via USB serial line
  - Use the Serial Monitor in the IDE
  - Or set up a C or Java (or you-name-it) interface
- Example Serial library calls
  - Serial.begin(*baud-rate*)
    - 9600 default
  - Serial.println(*string*)
  - int foo = Serial.read()
    - Read one byte (input data is buffered)
- See documentation for more

Example Program

```
int ledPin = 13;   // select the pin for the LED
int val = 0;       // variable to store the data from the serial port
void setup() {
  pinMode(ledPin,OUTPUT);    // declare the LED's pin as output
  Serial.begin(9600);        // connect to the serial port
}
void loop () {
  val = Serial.read();      // read the serial port
  // if the stored value is a single digit, blink the LED that number
  if (val > '0' && val <= '9' ) {
    val = val - '0';          // convert from character to number
    for(int i=0; i<val; i++) {
      Serial.println("blink!");
      digitalWrite(ledPin,HIGH);
      delay(150);
      digitalWrite(ledPin, LOW);
      delay(150);
    }
    //Serial.println();
  }
}
```