FYSIKUM

Digital System Construction

Lecture 7: Signal processing and CPUs

Digital signal processing (FIR, IIR)

Lab 8: digital notch filter

Processors

Embedded systems

Lab 6: Simple CPU in VHDL

Digital signal processing

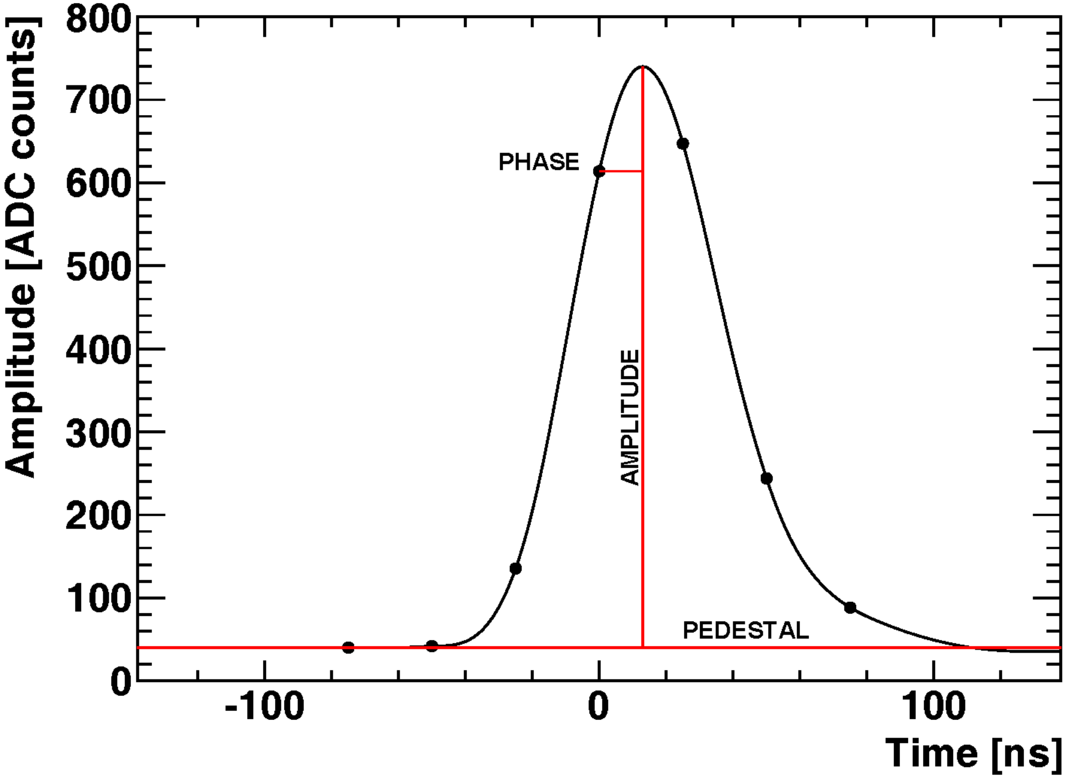
- Analyse/transform waveforms in discrete, digital format
  - Digital input data usually derived from analog signals using analog-digital converters (ADC)
  - Transformed output can be converted back to analog with digital-analog converters (DAC)
- DSP is a primary application of FPGAs
  - Telecoms are the biggest FPGA customers, so FPGAs designed with special “multiply-accumulate” DSP blocks
- Wide range of DSP techniques and applications
  - Can’t be covered in a single lecture
  - Will briefly present three common filter examples:
    - Finite Impulse Response (FIR) filter – pulse processing
    - Infinite Impulse Response (IIR) filter
    - Digital notch filter (FIR) (Lab 8)

Finite Impulse Response (FIR)

- Process data with finite length
  - Finite "window" of data points
  - For example: detector pulses measured in a few consecutive ADC samples
- FIR filter output is a weighted sum of the data points in the window
- Output is independent of data points outside of the window
  - No "memory" of earlier iterations
  - FIR filters are simple to implement and inherently stable

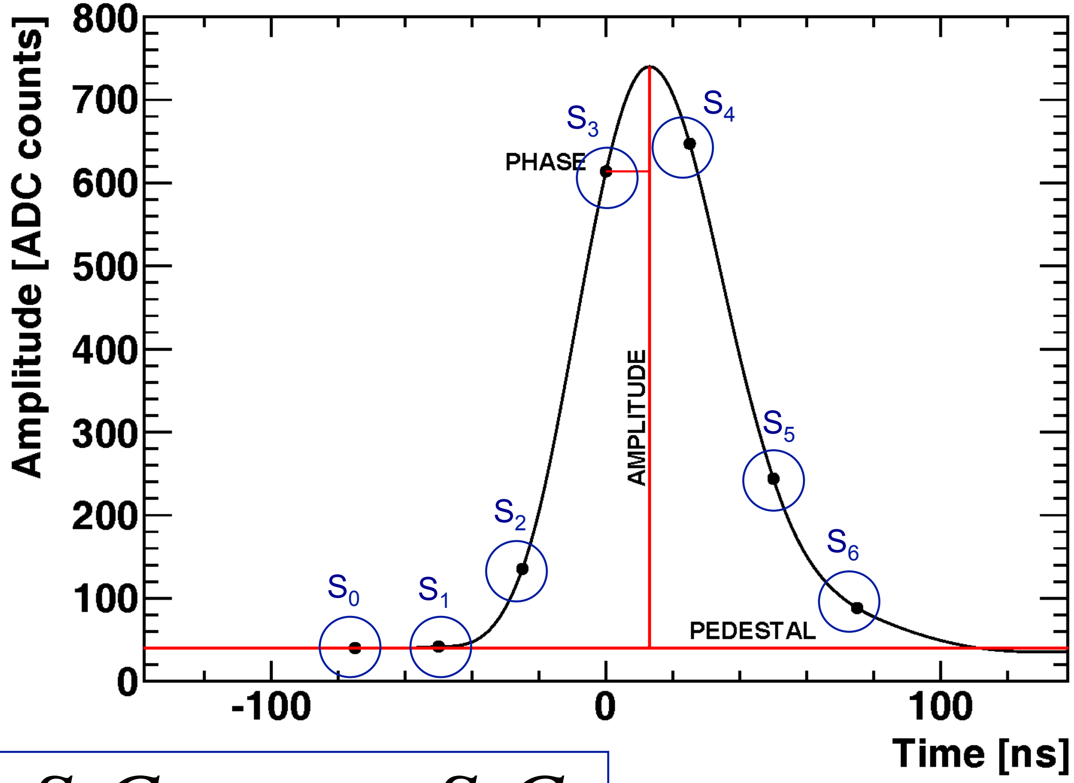
Example: detector pulse

- ATLAS hadronic Tile calorimeter
- Unipolar pulse (Only positive amplitude)
- Width ~150ns
- 25 MHz ADC rate (5-6 samples)



Applying a FIR filter

- “Sliding window” of 7 samples: S_n
- Each sample S_n multiplied by a coefficient C_n
- Sum products to produce the filter output:



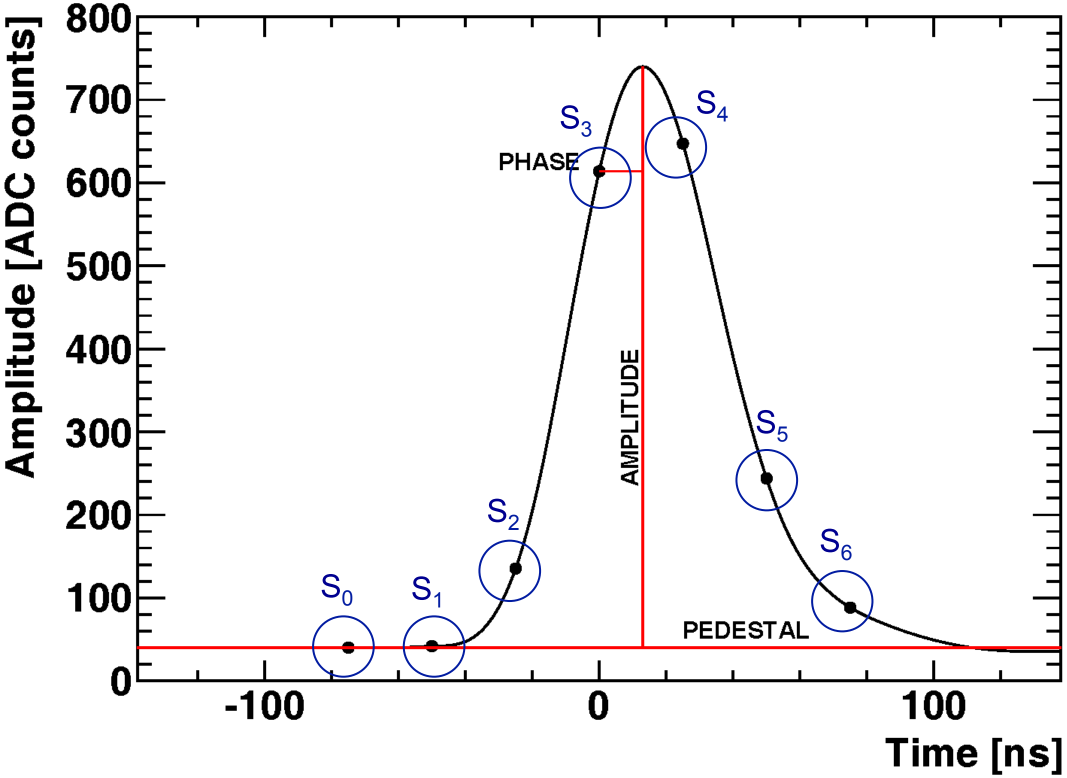
$$\boxed{F = S_0C_0 + S_1C_1 + S_2C_2 + \cdots + S_6C_6}$$

Coefficients C_n determine the filter response

Example: pulse integral

- Samples have equal weights
  - "Area under the curve"
- Filter coefficients C_n:

Sample	Weight
S_0	0
S_1	1
S_2	1
S_3	1
S_4	1
S_5	1
S_6	1

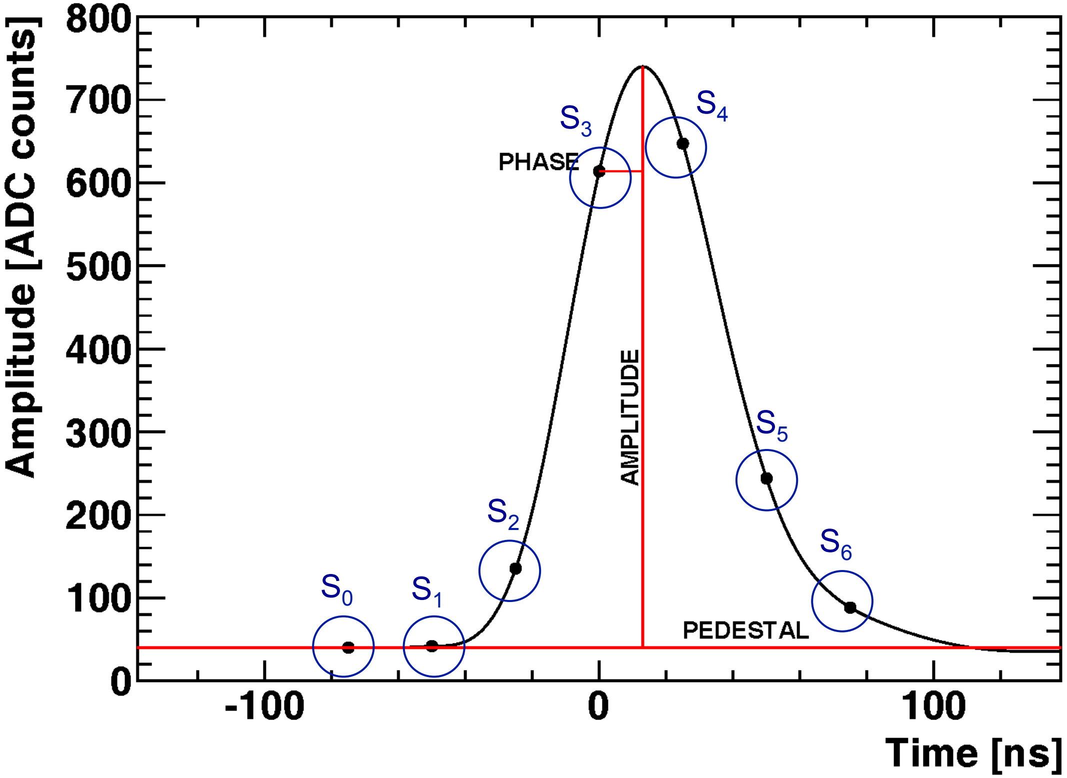


$$F = S_1 + S_2 + \cdots + S_6$$

Integral with pedestal subtraction

- Subtract pedestal (S_0) from the other six samples.
- Filter coefficients C_n:

Sample	Weight
S_0	-6
S_1	1
S_2	1
S_3	1
S_4	1
S_5	1
S_6	1



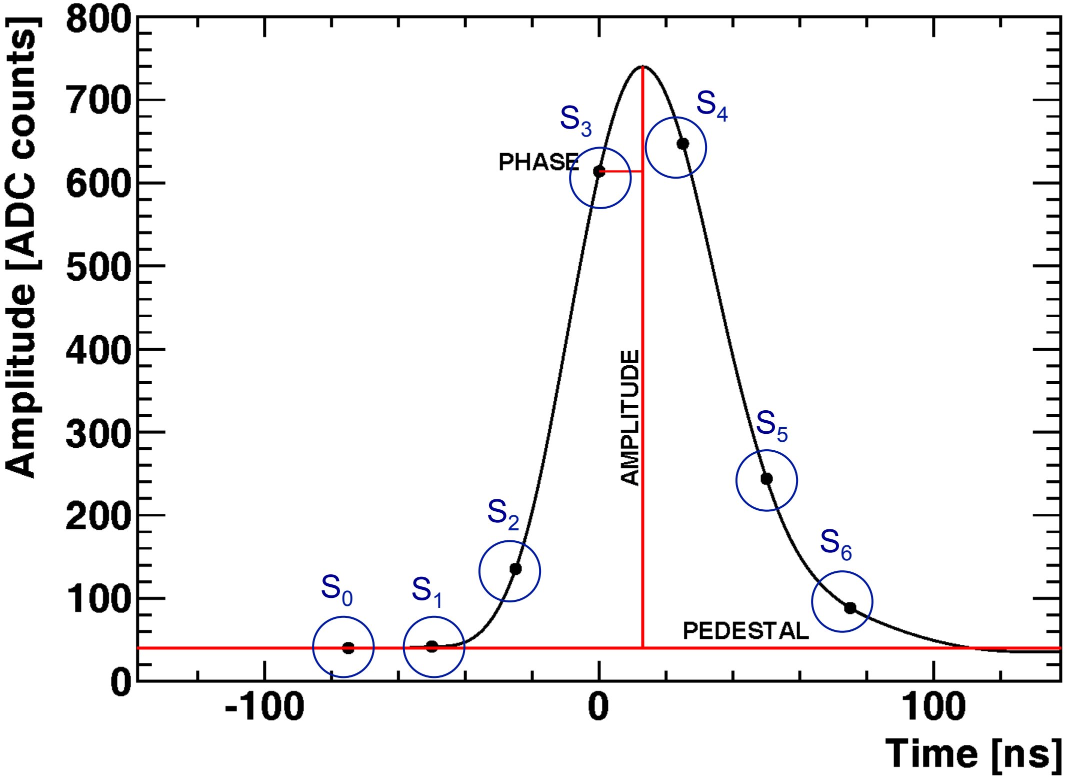
$$F = (S_1 + S_2 + \cdots + S_6) - (6 \times S_0)$$

Peak amplitude

- Use only the maximum sample
  - For example, S_4
- Filter coefficients C_n:

Sample	Weight
S_0	0
S_1	0
S_2	0
S_3	0
S_4	1
S_5	0
S_6	0

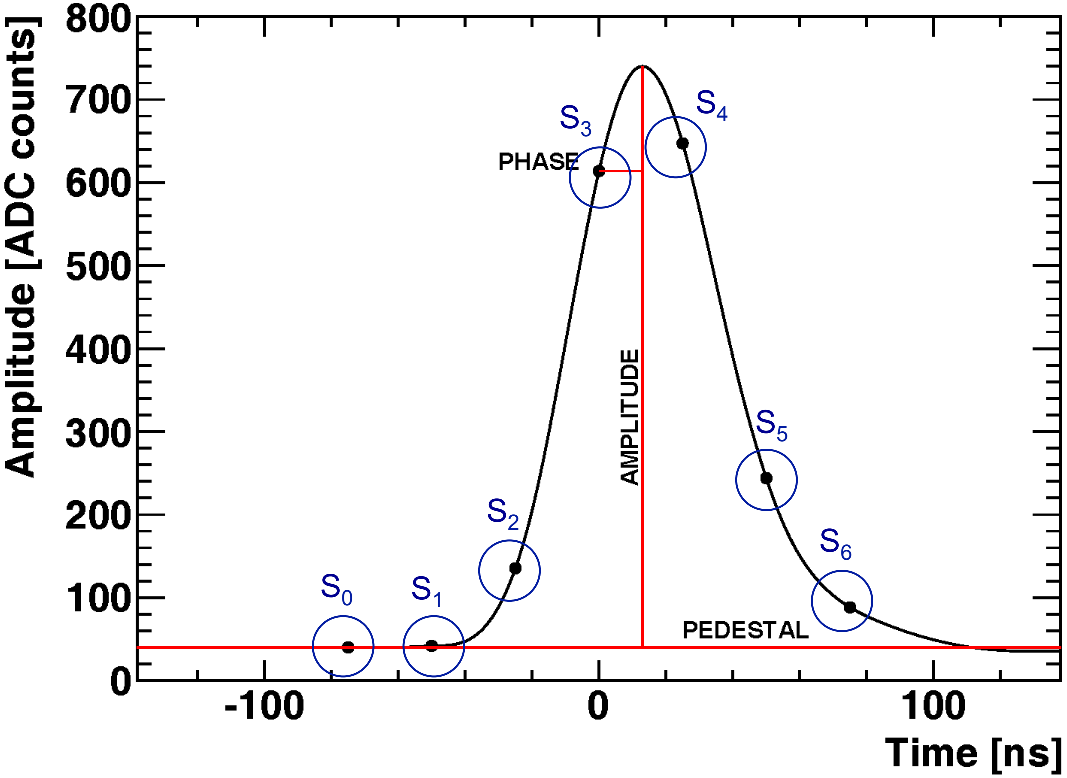
$$\boxed{F = S_4}$$



“Optimal filter”

- Match coefficients to the ideal pulse shape (with noise)
- Gives best resolution and signal/noise performance
- Filter coefficients C_n:

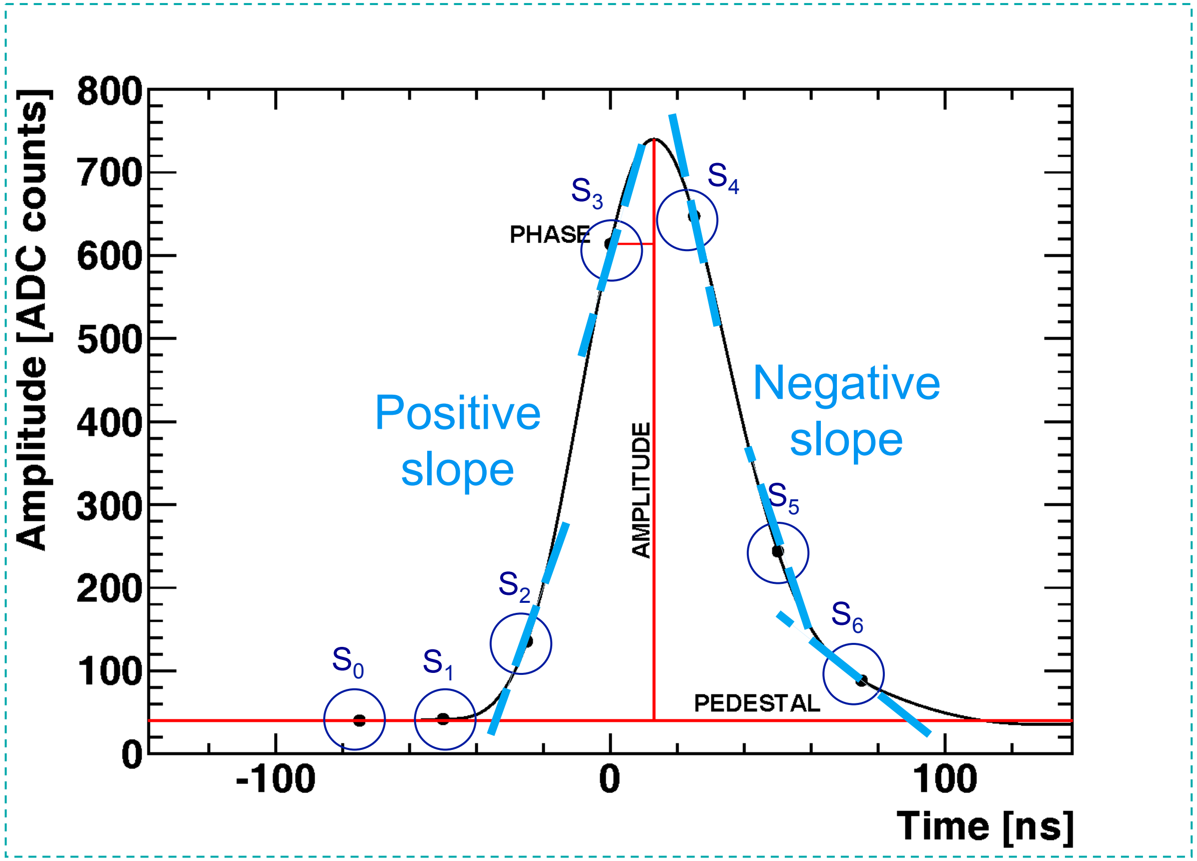
Sample	Weight
S_0	-172
S_1	0
S_2	14
S_3	62
S_4	64
S_5	24
S_6	8



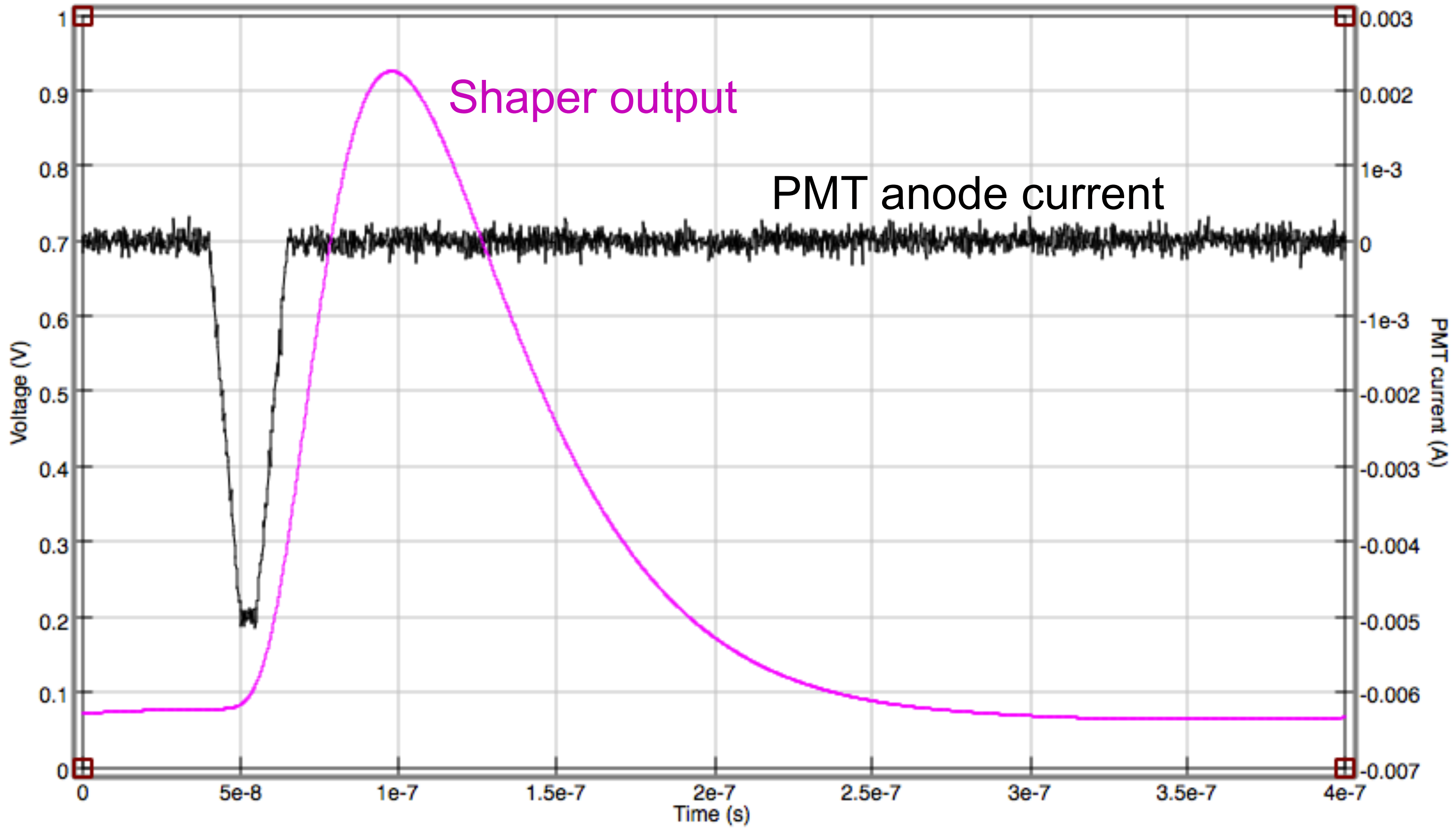
Maximum response when samples are aligned with the ideal pulse shape

Timing (phase) measurement

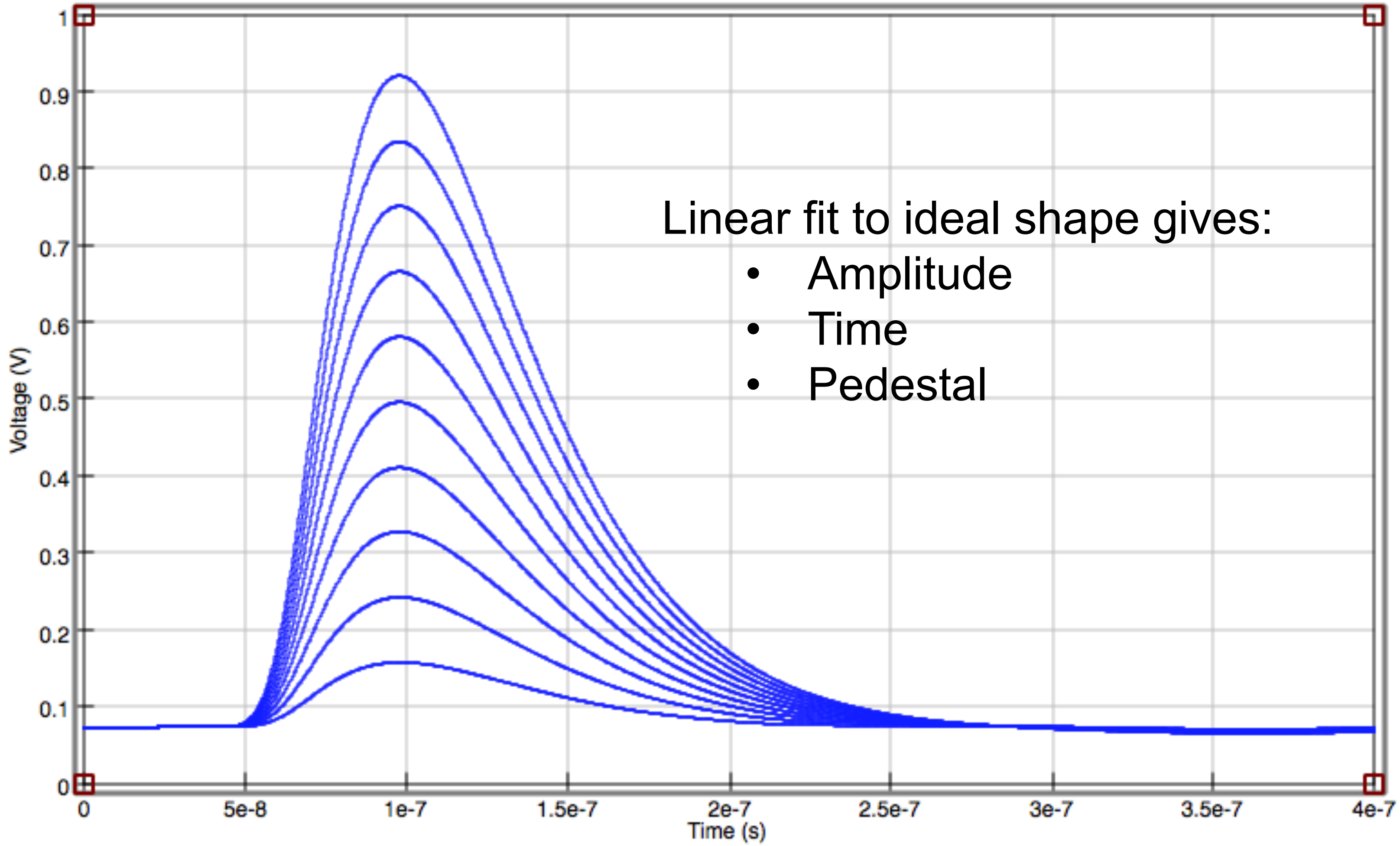
- Use derivatives of ideal curve as coefficients
- Divide the timing filter output by the pulse amplitude
- Can achieve sub-ns timing resolution with 25 ns sampling rate



Simulated PMT pulse shaper



Shape is amplitude independent

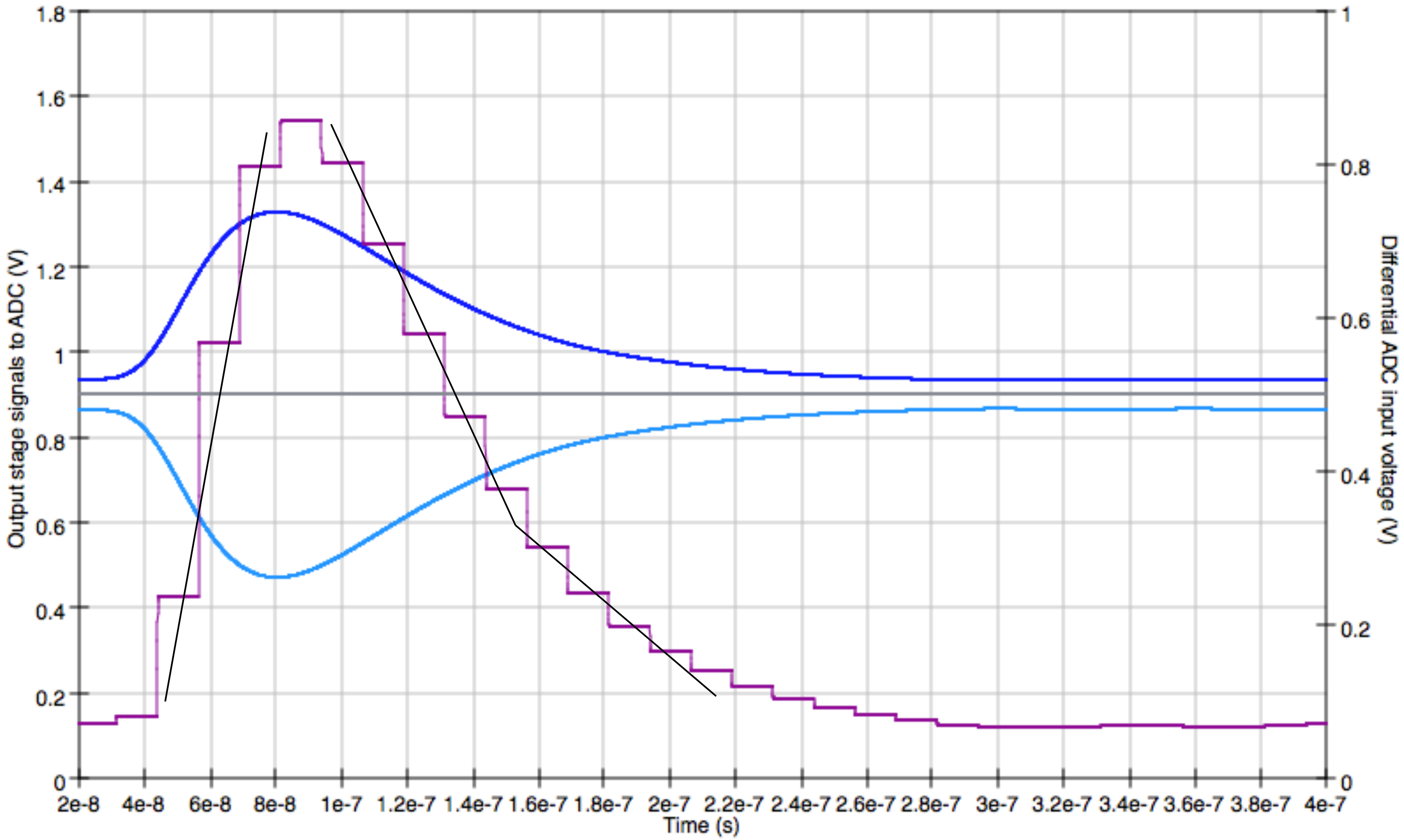


Linear fit to ideal shape gives:

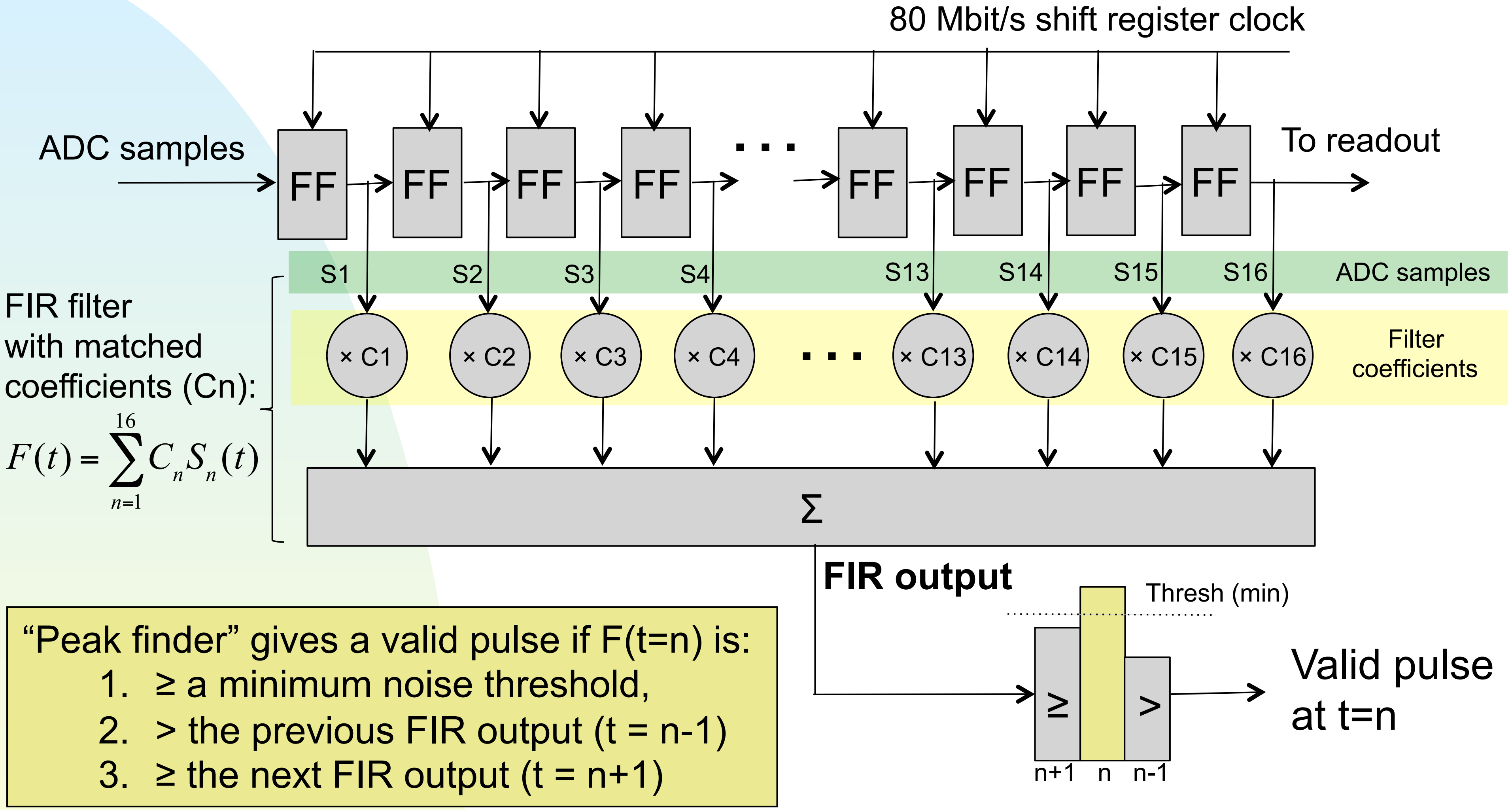
- Amplitude
- Time
- Pedestal

Sampled shape (80 MHz)

Multiple samples on rising and falling edges to allow sub-nanosecond pulse timing reconstruction



FIR filter architecture



FIR filter with matched coefficients (Cn):

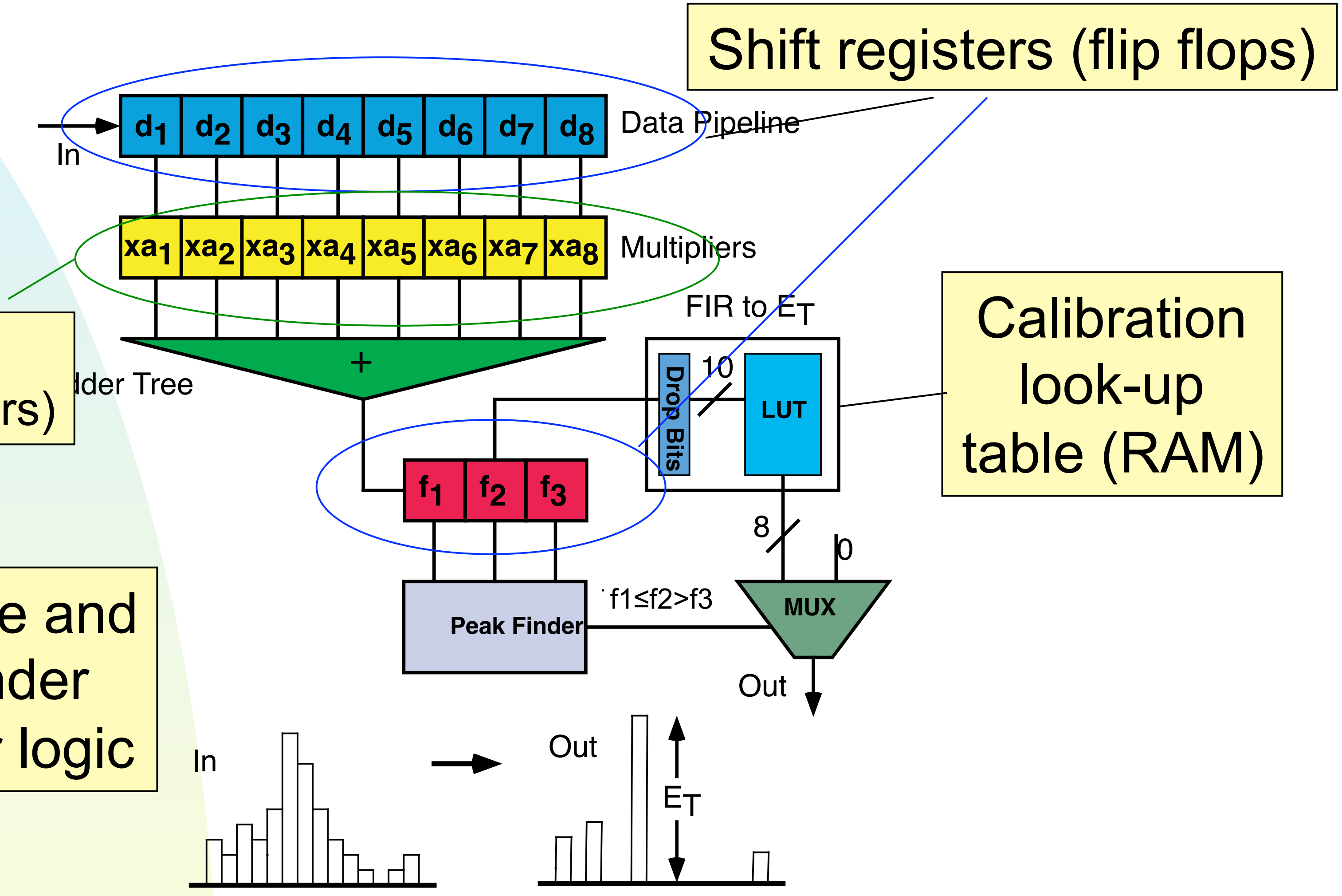
$$F(t) = \sum_{n=1}^{16} C_n S_n(t)$$

"Peak finder" gives a valid pulse if F(t=n) is:

1. ≥ a minimum noise threshold,
2. > the previous FIR output (t = n-1)
3. ≥ the next FIR output (t = n+1)

Similar to ATLAS L1Calo bunch-crossing ID algorithm

ATLAS L1Calo PreProcessor



Shift registers (flip flops)

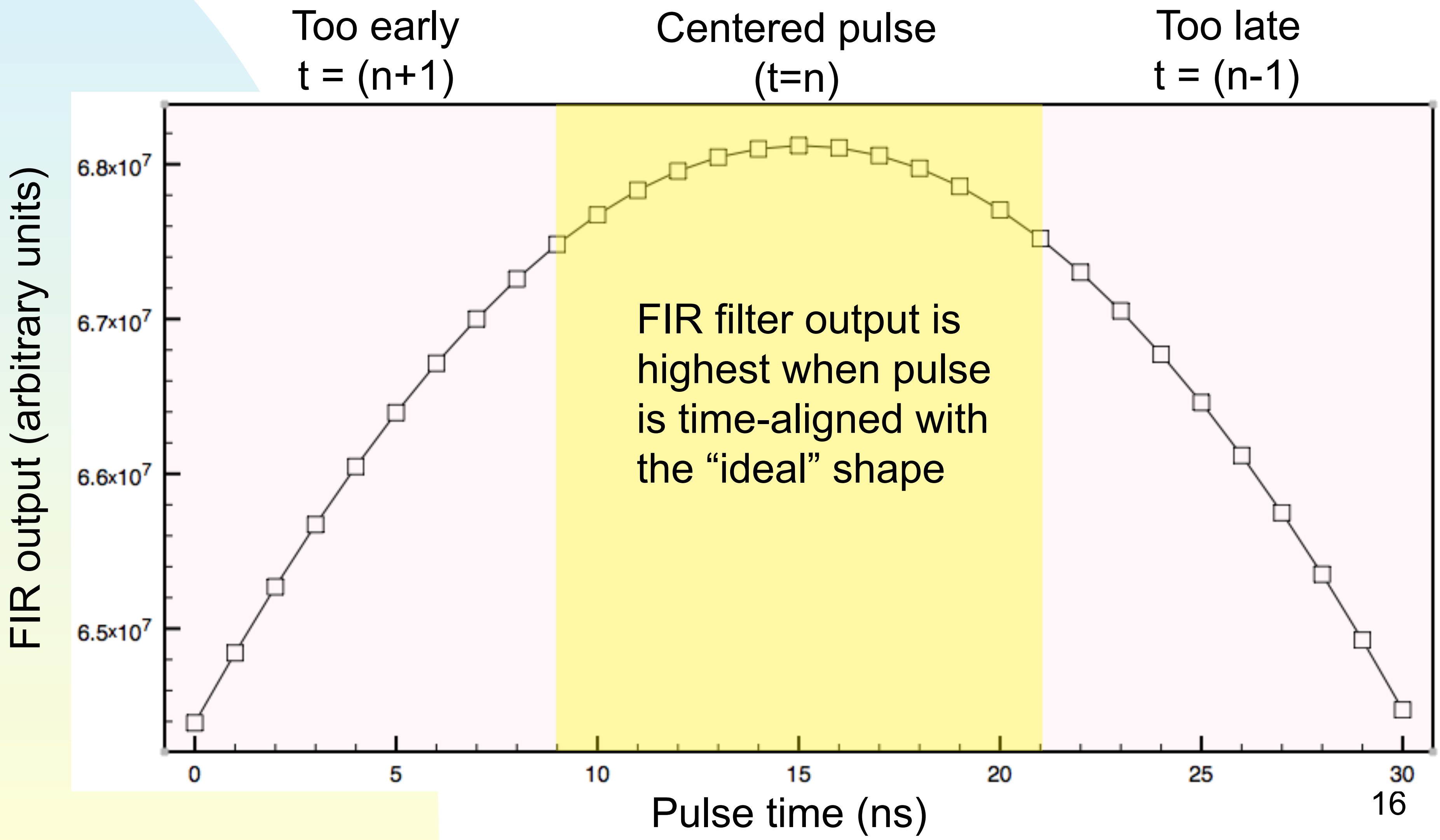
FIR filter (DSP multipliers)

Calibration look-up table (RAM)

Adder tree and peak finder in regular logic

FIR filter amplitude output

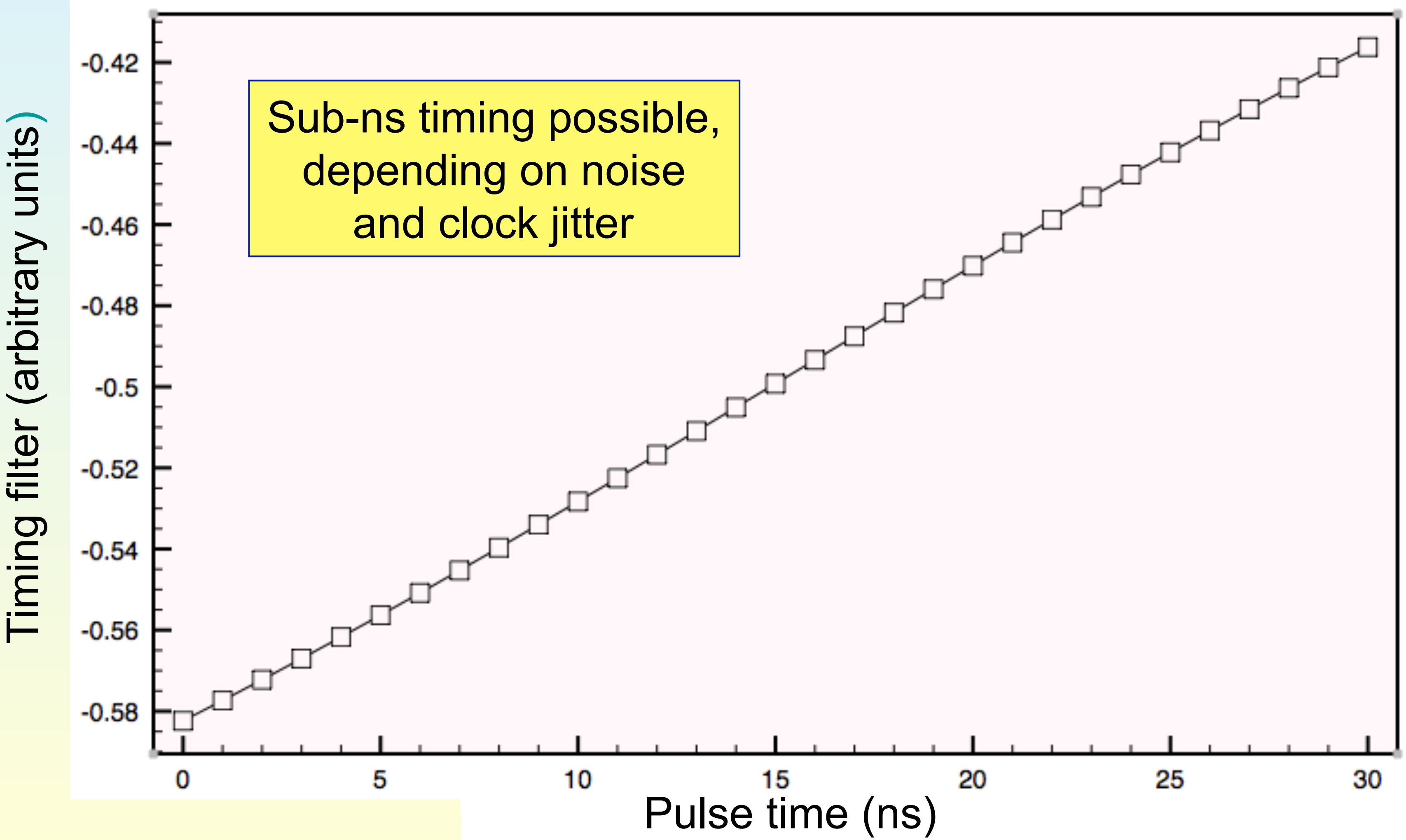
(15 samples)



FIR filter output is highest when pulse is time-aligned with the "ideal" shape

Timing filter output

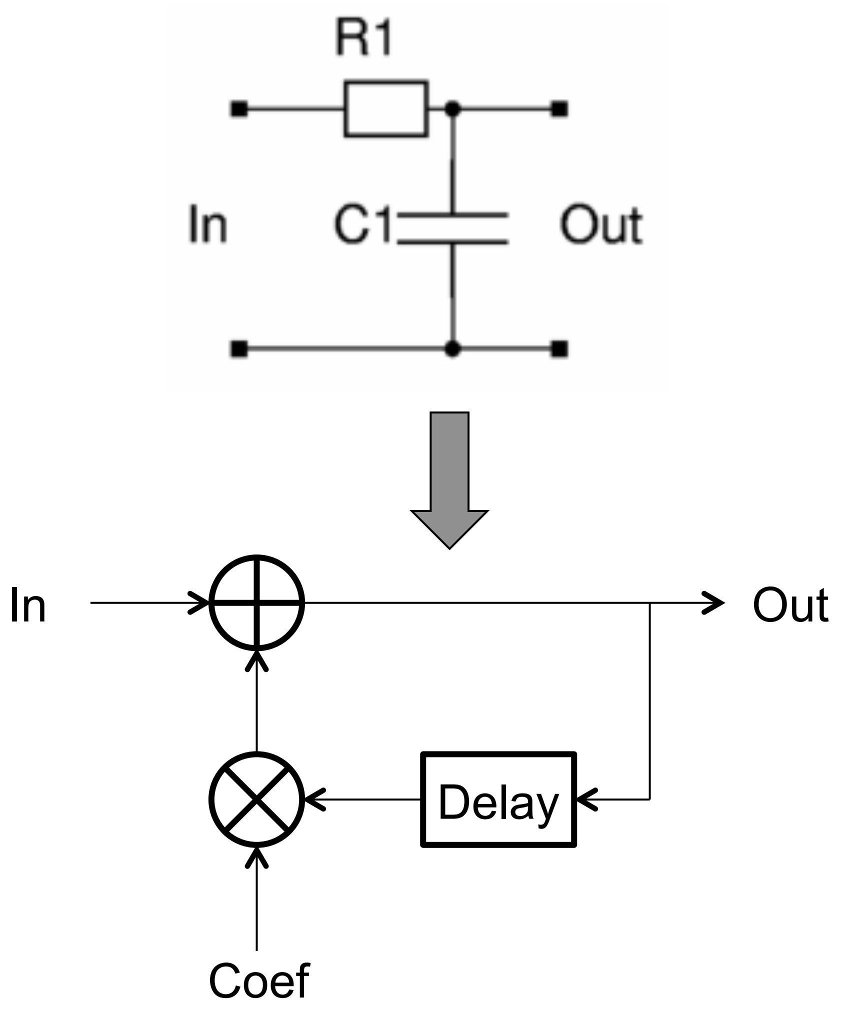
(FIR coefficients ~ derivatives of pulse shape)



Sub-ns timing possible, depending on noise and clock jitter

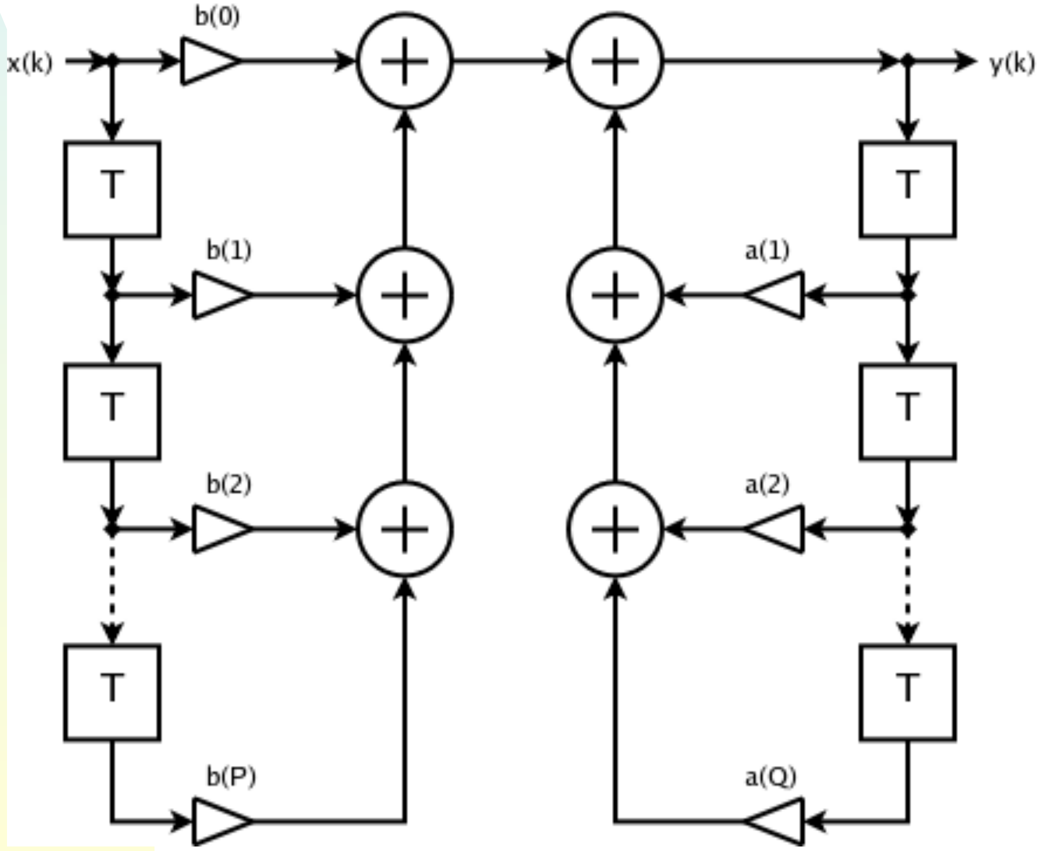
Infinite impulse response (IIR)

- Recursive filter
  - Output depends on on previous history
  - Delay provides a “memory” of previous states (like C in the analog filter)
- Efficient to implement, but not inherently stable like FIR



General form of an IIR

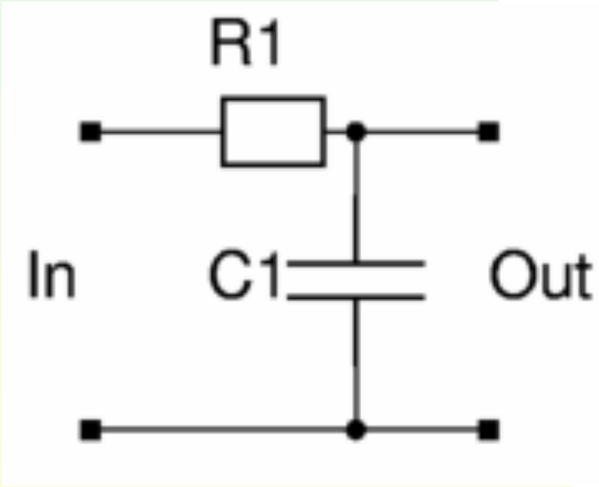
$$
\begin{aligned}
y_n &= b_0 x_n + b_1 x_{n-1} + b_1 x_{n-1} + \cdots \\
&\quad + a_1 y_{n-1} + a_2 y_{n-2} + \cdots
\end{aligned}
$$

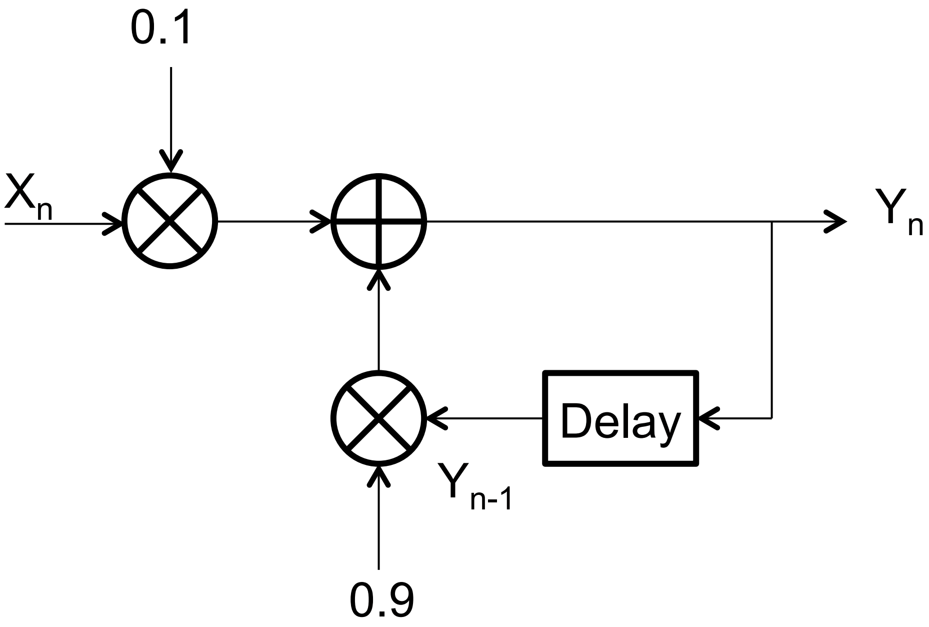


Simple low-pass IIR filter

y ≅ the integral of x
with a slow decay

$$y_n = \overset{b_0}{\overset{\downarrow}{0.1}} \cdot x_n + \overset{a_1}{\overset{\downarrow}{0.9}} \cdot y_{n-1}$$

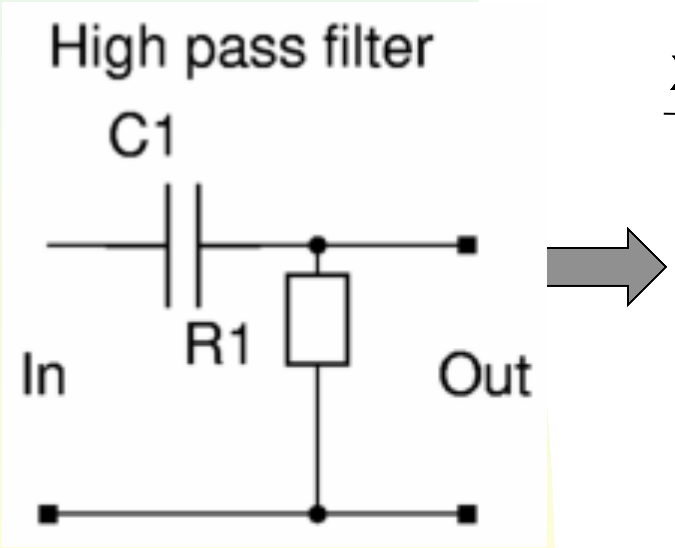


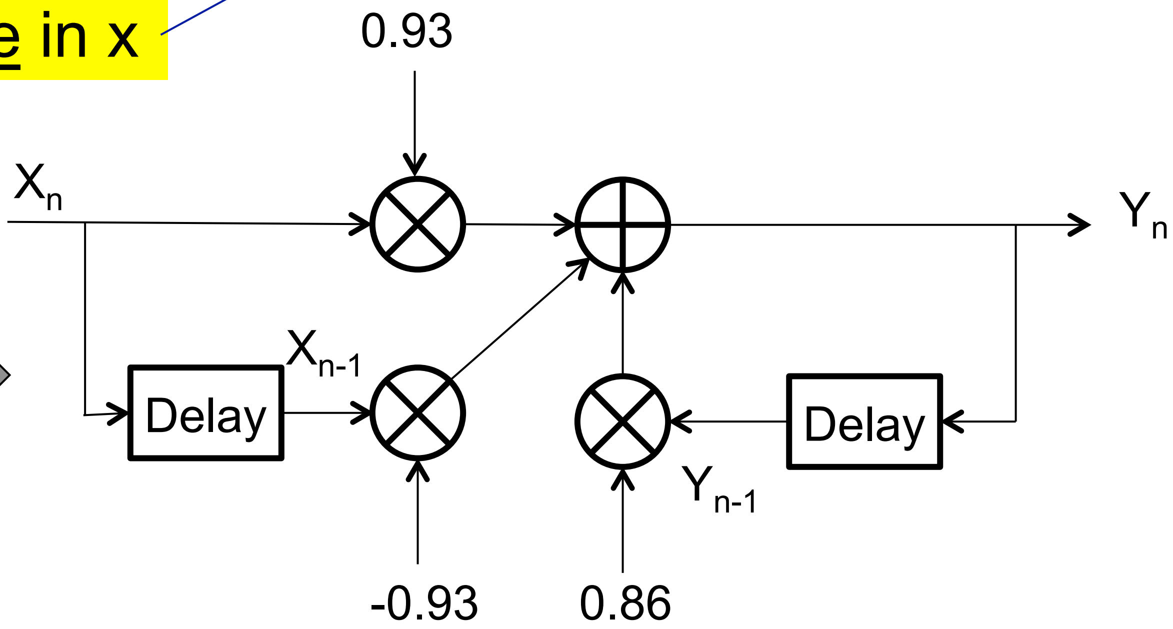


Simple high-pass IIR filter

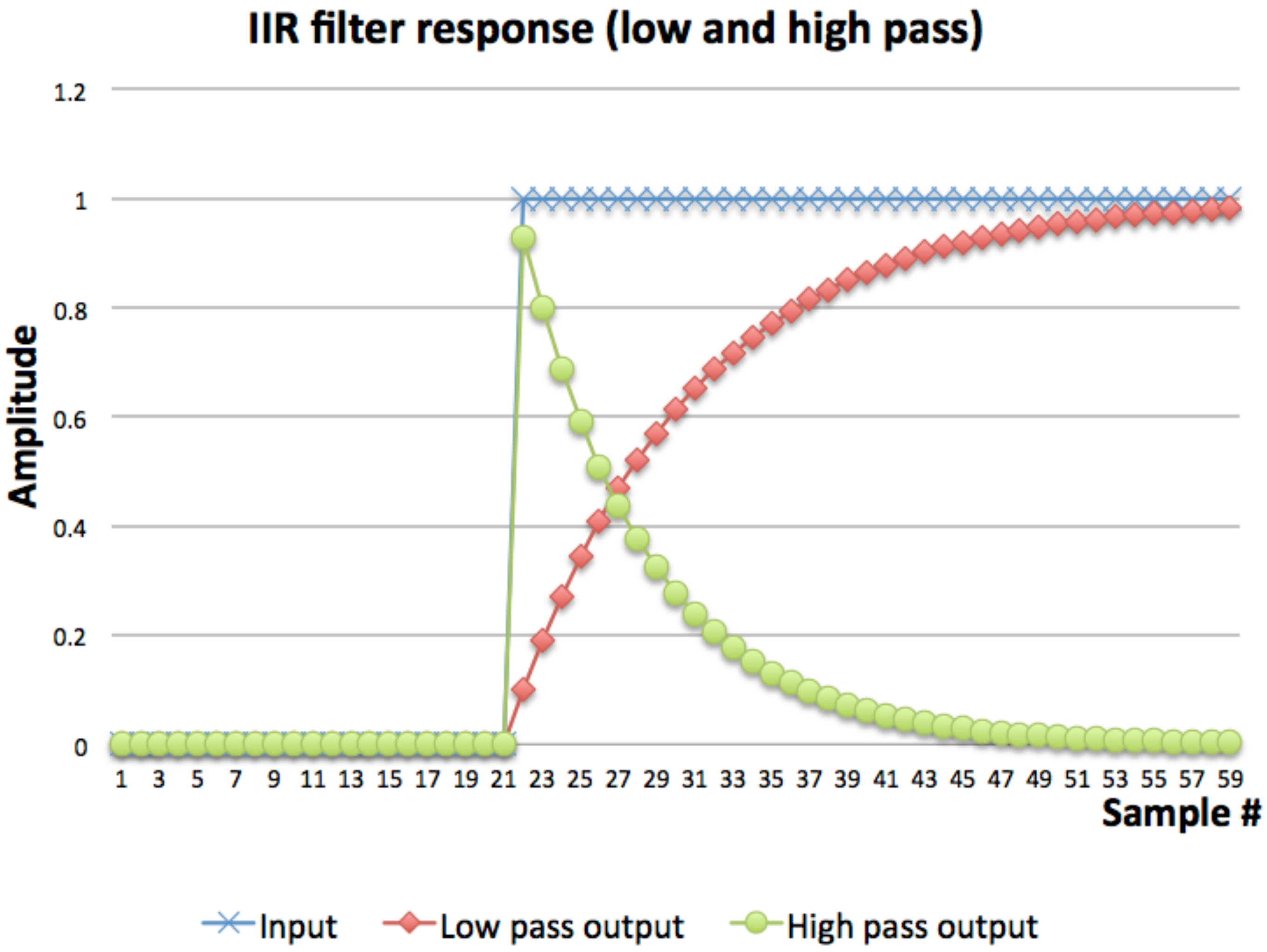
$$\begin{aligned} y_n &= \overset{\overset{b_0}{\downarrow}}{0.93} \cdot x_n - \overset{\overset{b_1}{\downarrow}}{0.93} \cdot x_{n-1} + \overset{\overset{a_1}{\downarrow}}{0.86} \cdot y_{n-1} \\ &= 0.93 \cdot (x_n - x_{n-1}) + 0.86 \cdot y_{n-1} \end{aligned}$$

y integrates the <u>change</u> in x



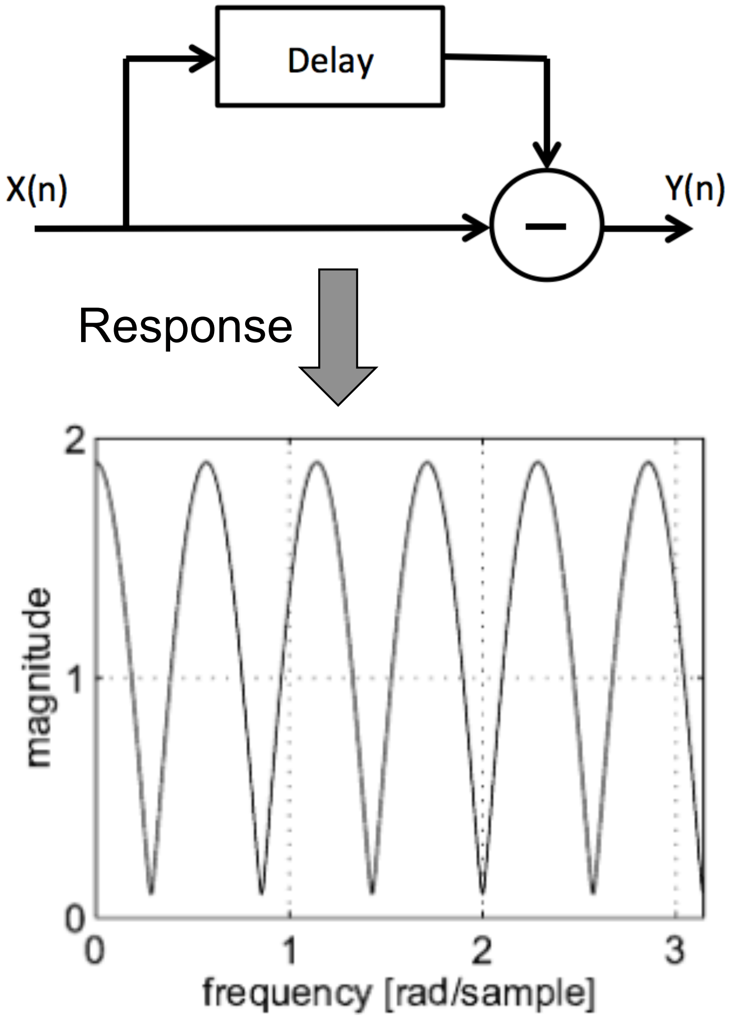


Filter response (time domain)



Lab 8: Digital notch filter (FIR)

- Simple FIR filter to block a given frequency and all of its higher harmonics
  - AKA “comb” filter
- Delay sets the base frequency
  - $F_B = \text{Delay}^{-1}$
- Use ADC and DAC to filter analog signals in real time

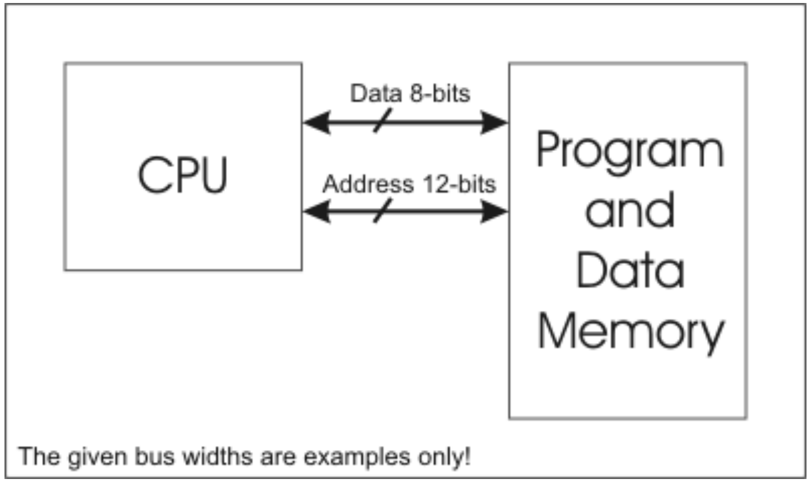


Basic processor elements

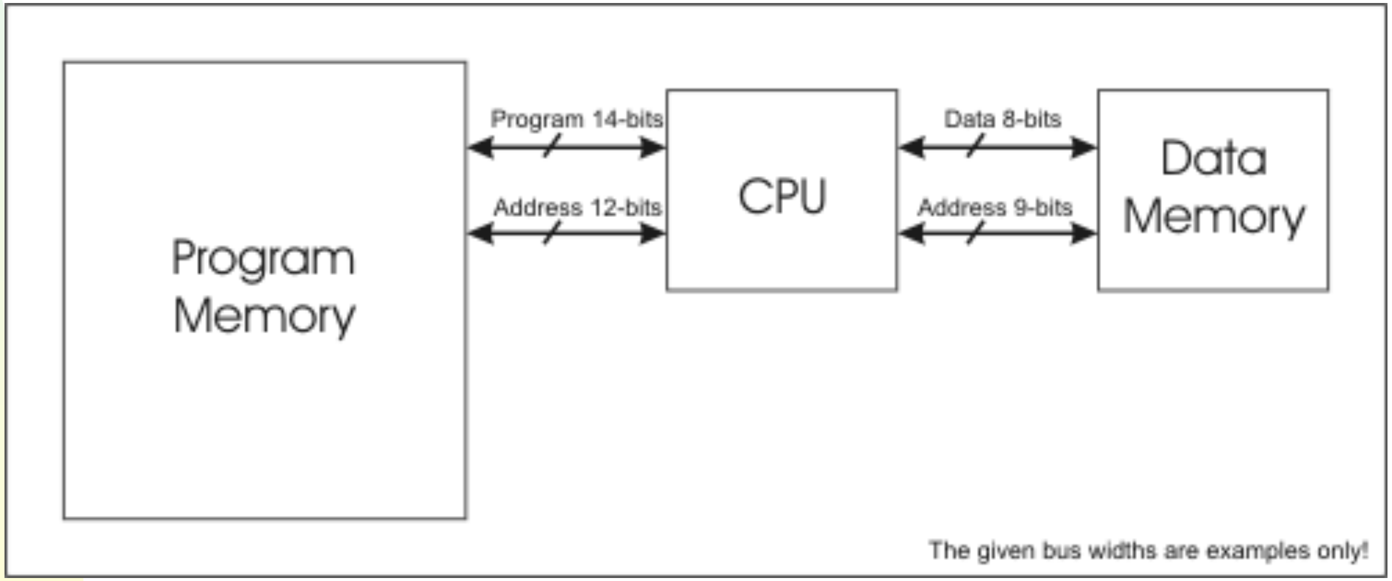
- Processor
  - State machine for executing commands
  - Logic unit(s) for different operations
  - Registers for control, temporary storage
- Memory interface(s)
  - Access and store instructions, data
- Peripheral interface(s)
  - Interfaces with the rest of the world

Two basic architectures:

Von Neumann



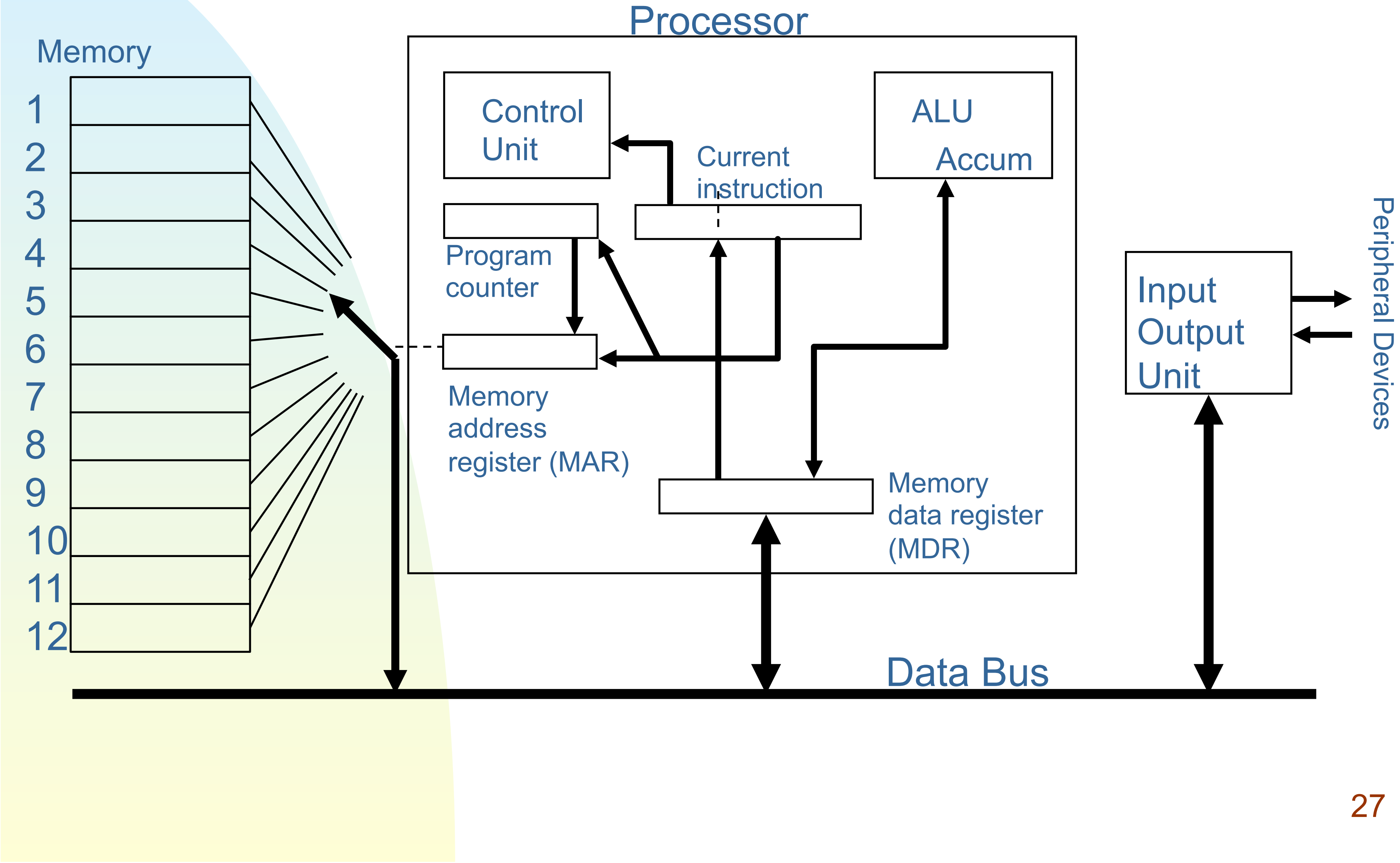
Harvard



Components of a CPU

- Control unit
  - Decode and execute instructions
- Arithmetic and Logic Unit (ALU)
  - Perform math and logic algorithms
- Registers:
  - Store temporary data
  - Control and/or monitor specific functions

Simple Von Neumann CPU



Basic CPU functionality

- Fetch instruction (from memory)
- Interpret instruction
  (determine what action to perform)
- Fetch data (get additional operands from memory or registers as needed)
- Execute the instruction
- Write data (store results in memory or registers)

Instruction sets

- CPUs are designed with a list of instructions that it can execute (instruction set)
- Each instruction has an identifying code (**opcode**)
- Instructions usually require additional information to perform a task. Some examples:
  - ADD, ASSIGN, MOV, JMP, EXIT
- Most opcodes need additional data (**operands**) in order to be executed:

Addresses	(location of data/next instruction/etc)
Numbers	(integer/float/BCD)
Characters	(ASCII / EBCDIC)
Logical data	(boolean/status/etc)

High-level vs low-level code

- **C code**

```
int x=0 ;
int y=0 ;
main
{
x=2 ;
y=3 ;
x=x+y;
}
```

Assembly code

```
.model small
.stack 100h
.data
x   dw   0
y   dw   0
.code
mov x, 2
mov y,  3
mov ax,  x
add ax,  y
mov x, ax
end.
```

CPU registers

- CPUs use registers to:
  - Hold instructions to be executed
  - Hold data being processed
  - Hold memory address(es)
  - Control and monitory CPU functionality and/or peripherals
  - Etc.
- Some visible to programmer, others only only by CPU and special O/S functions.
- May be general purpose, or special purpose.

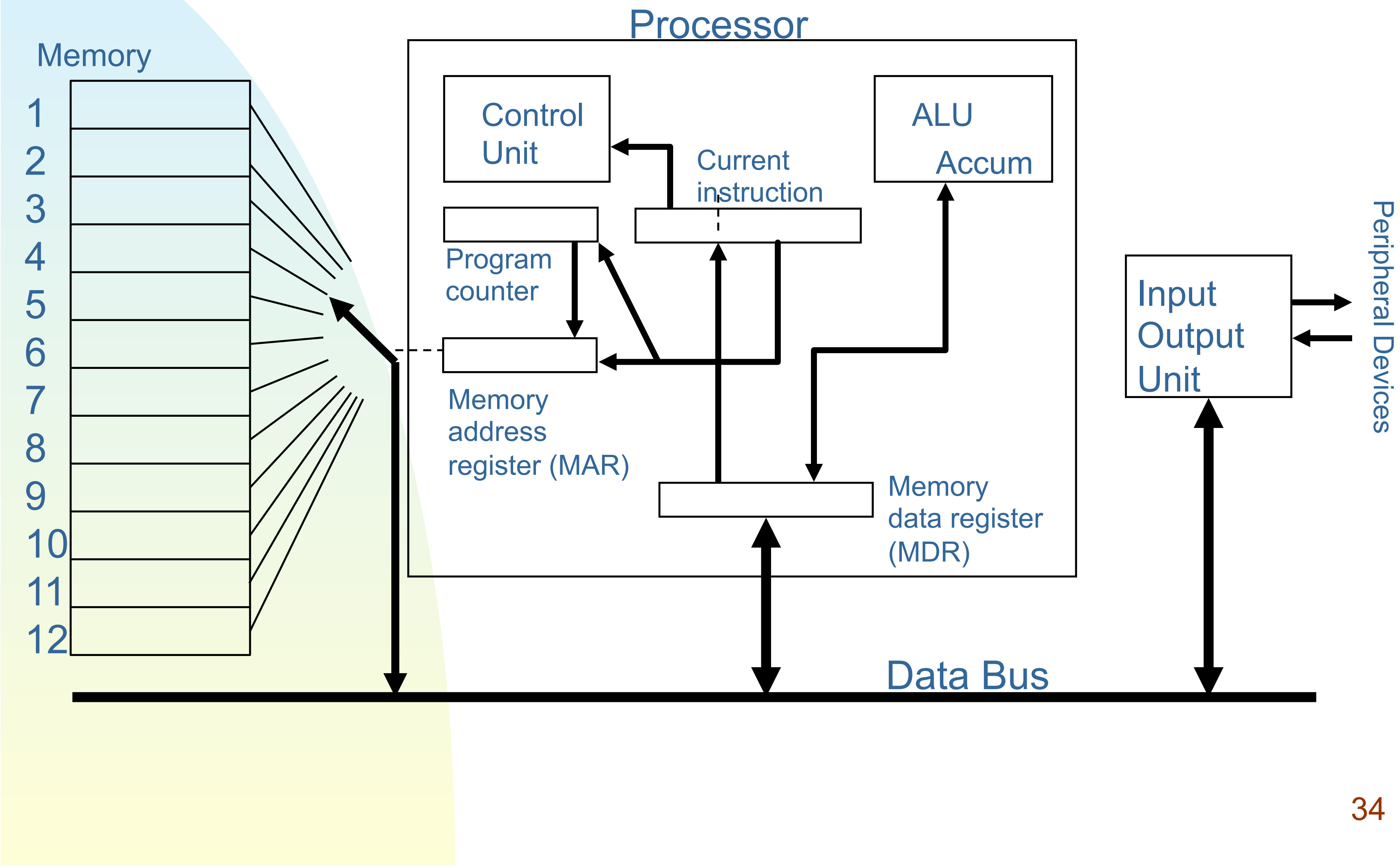
General purpose registers

- General purpose registers:
  - Visible to the user/programmer
  - Usable by many operations
    - E.g. temporarily hold an address (e.g. pointer) or data (intermediate result)
  - Early CPUs had one: the accumulator
    - Hold intermediate results of a calculation

Registers

- Early CPUs had one general-purpose register
  - Accumulator (hold intermediate results)
- Special purpose registers include:
  - Program Counter (PC)
    - Address of next instruction to be fetched from memory
  - Current Instruction Register (CIR)
    - Current instruction being executed.
  - Memory address and data registers (MAR & MDR)
    - Used for reading/writing to memory
- Registers can also: control/monitor peripherals, error/status flags, etc.

Von Neumann CPU (review)



Computer system types

- General-purpose
  - PCs, mainframes, etc
- Embedded systems
  - Microcontrollers
  - Digital signal processors
  - etc.

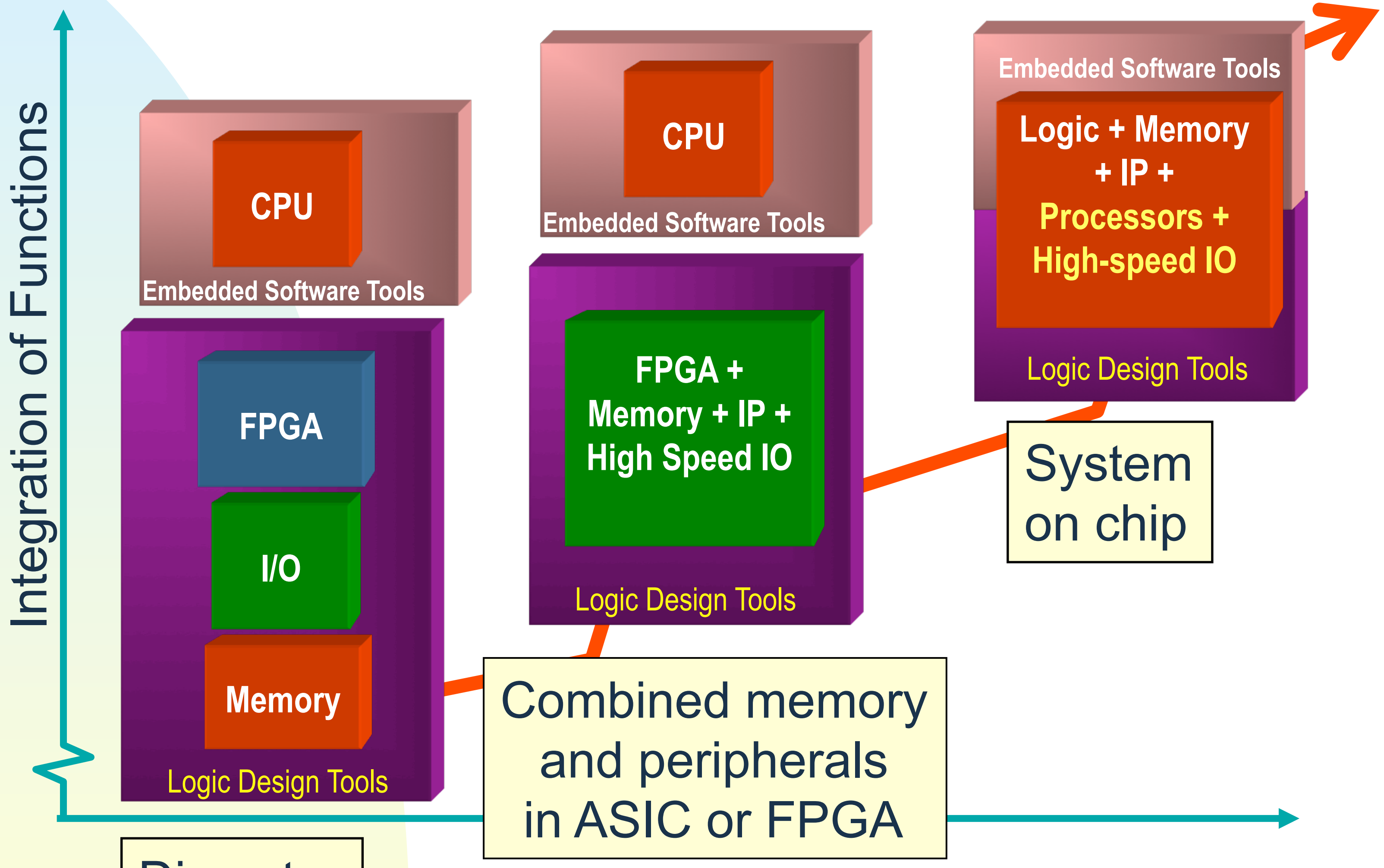
Embedded Systems

- Nearly any computer with the following properties:
  - Single function
    - Dedicated to running a specific application
  - Tightly constrained, minimal architecture
    - Low cost; single-to-few components
    - Works 'fast enough'
    - Low power (especially for portable devices)
  - Reactive, real-time operation
    - Continually monitors environment, and reacts to changes
  - Hardware and software co-exist
    - 'Hardware-level' programming

embedded application examples

- Communication devices
  - Network routers and switches
  - Smart phones
- Automotive
  - Braking systems, traction control, airbag release systems, cruise-control, fuel injection, etc...
- Aerospace
  - Flight-control systems, engine controllers, autopilots, passenger in-flight entertainment
- Measurement systems
  - Data acquisition hardware, slow-control and monitoring, user interfaces (buttons and touch screens), etc.

Different levels of integration

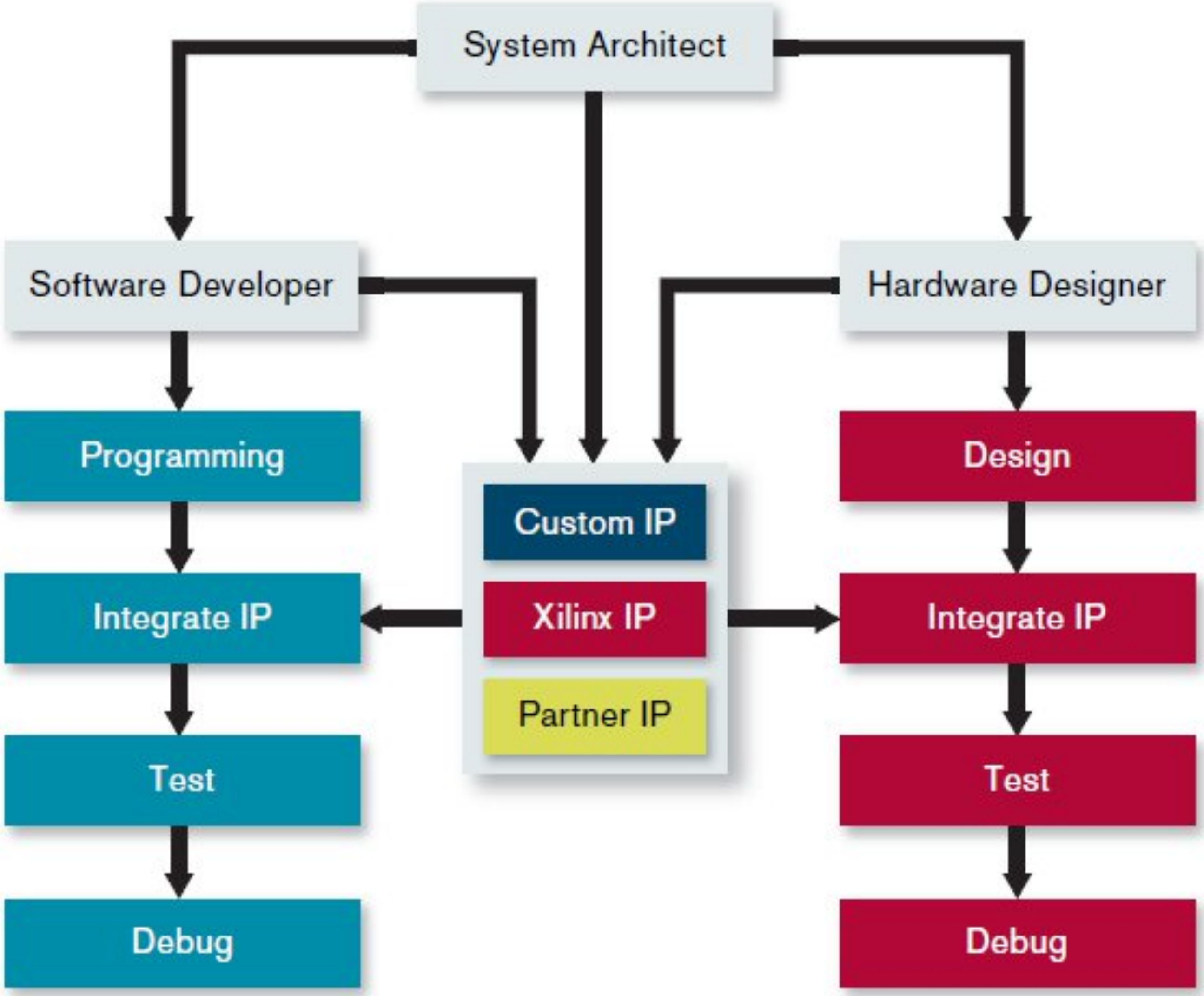


System on chip

Combined memory and peripherals in ASIC or FPGA

Discrete elements

FPGA Embedded development



Microcontrollers

- Dedicated single-chip system
  - Integrated CPU, memory, peripherals
- Designed for real-time measurement, communication
  - Integrated analog-digital conversion
  - UART, I2C, SPI, etc...
- Low-cost, low-power architectures
  - Limited CPU speed (MHz range)
  - Limited memory resources
  - Small package with limited pins

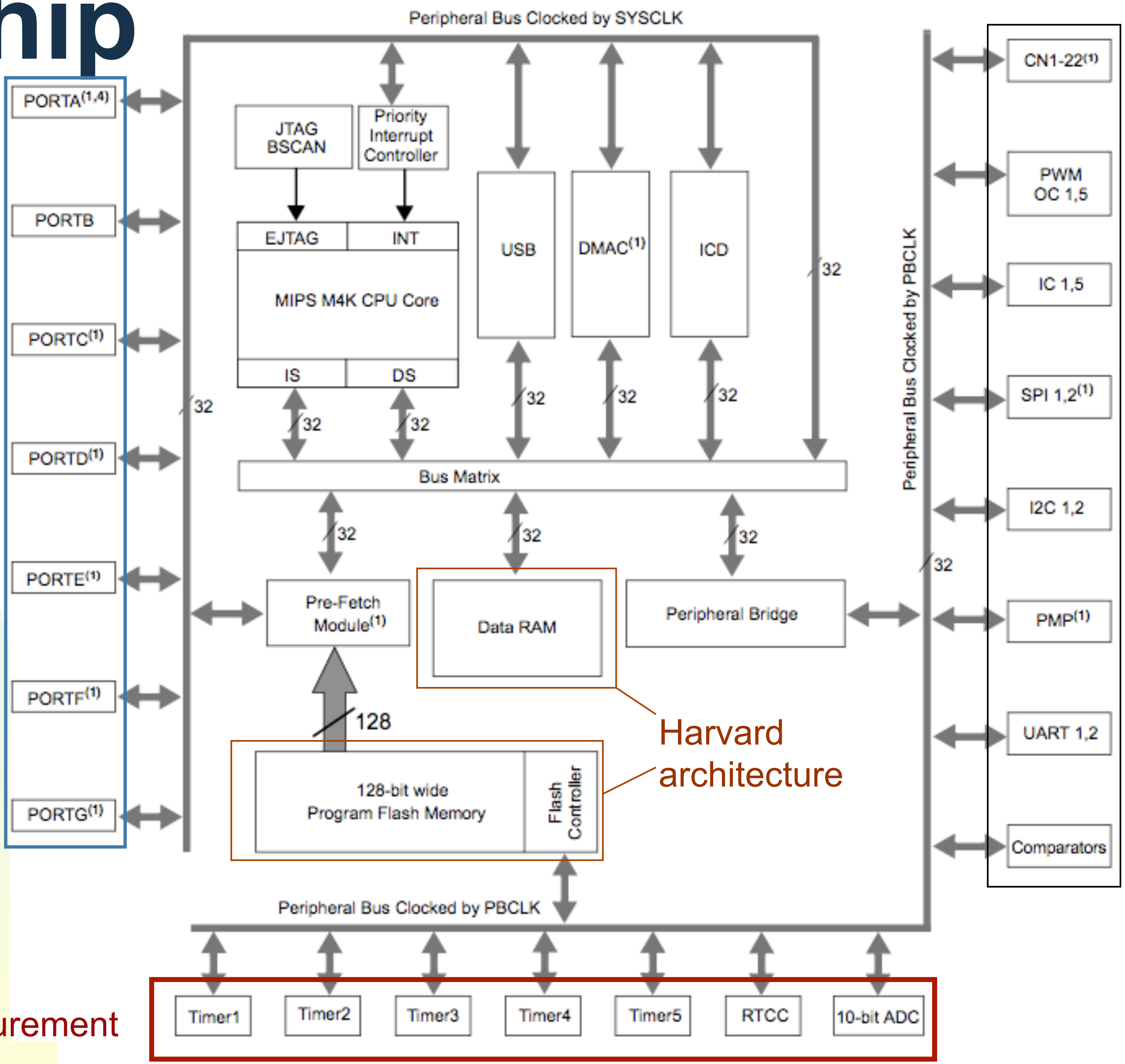
Microchip PIC32

Communication, interrupts, etc

General-use digital I/O

Harvard architecture

Measurement



Register-based control

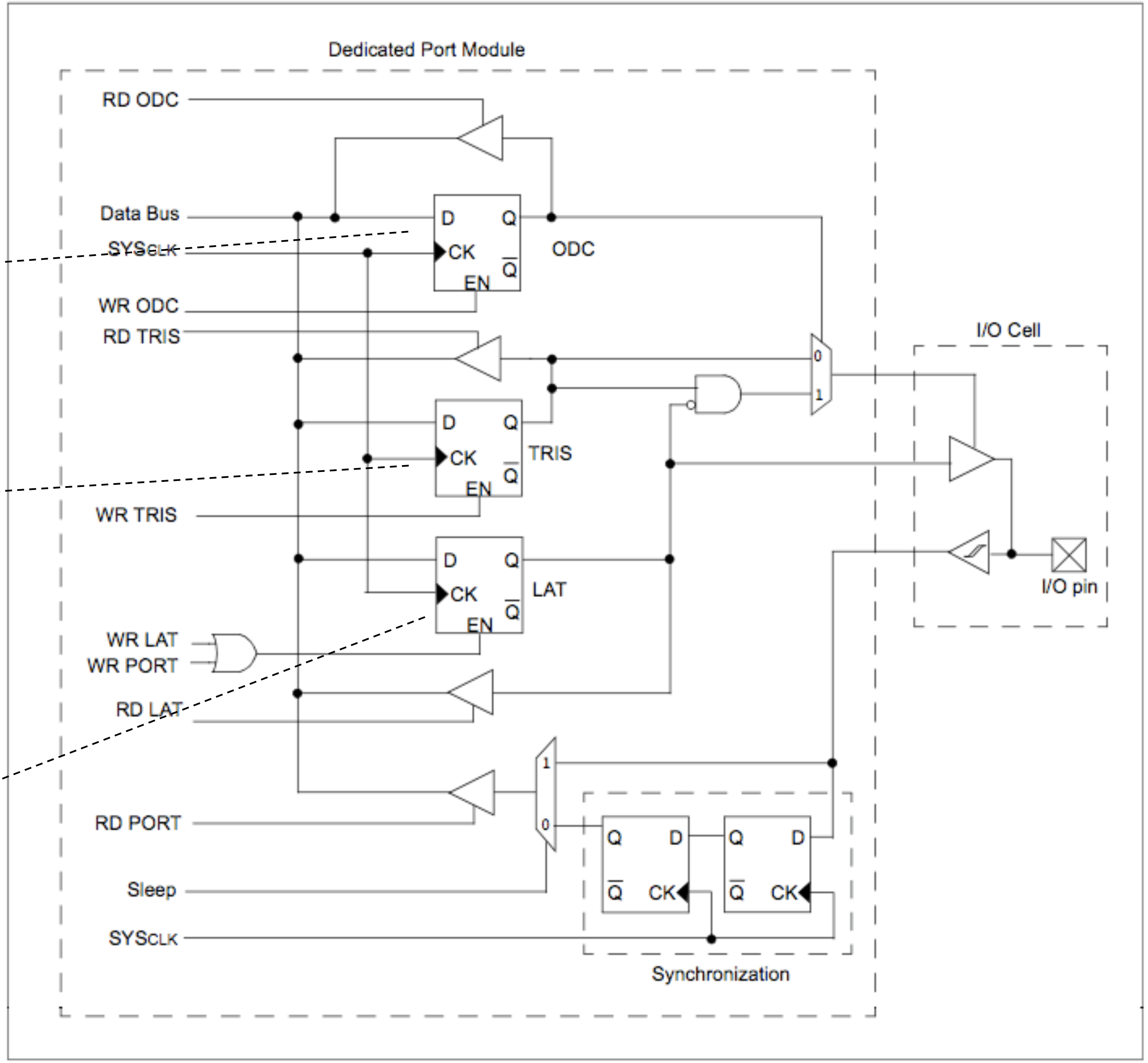
Example:
Digital
I/O port

Open-drain control (ODC)

Tri-state (TRIS)

Register for output data

FIGURE 12-1: BLOCK DIAGRAM OF A TYPICAL PORT STRUCTURE



Register control of I/O ports

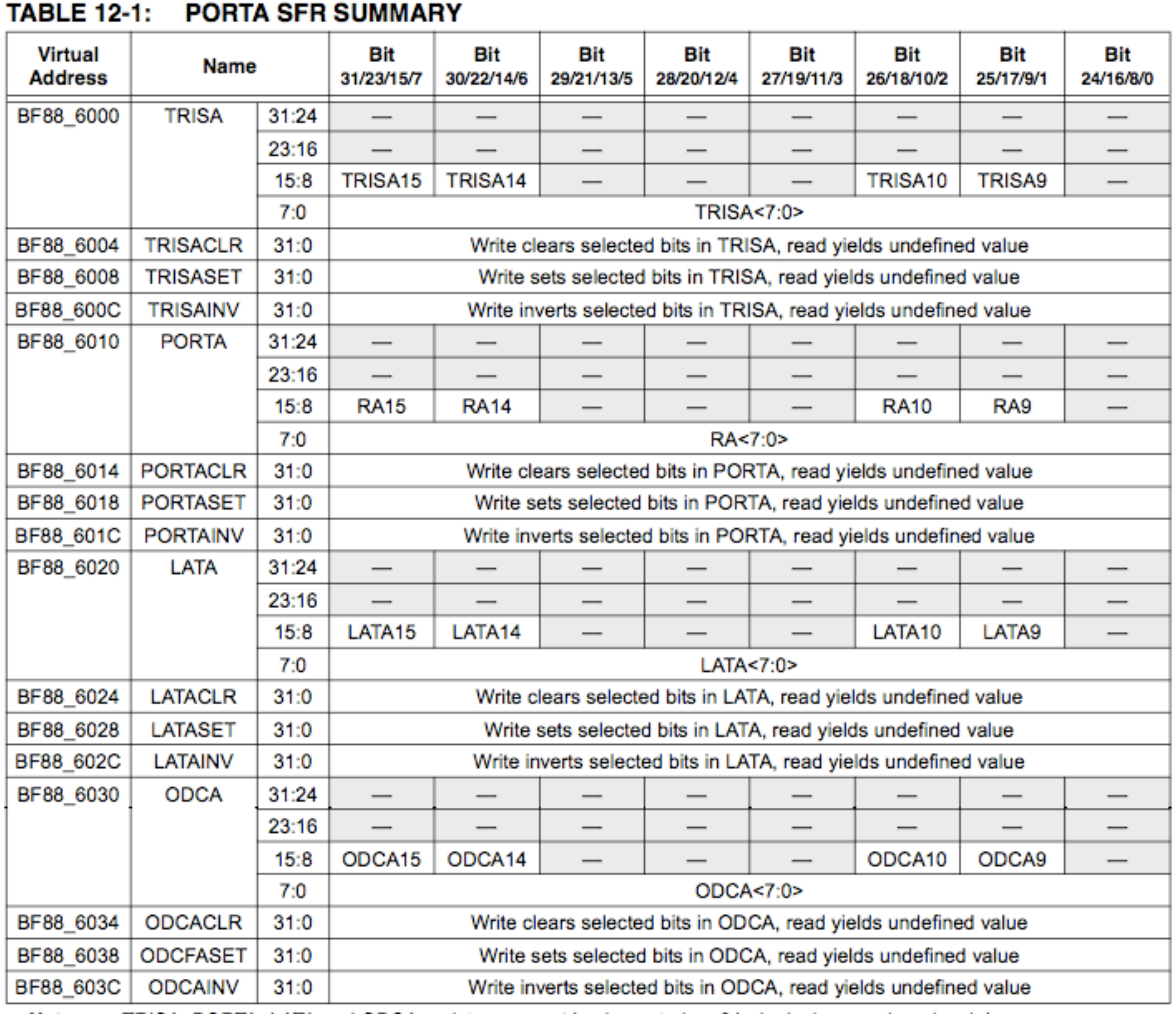
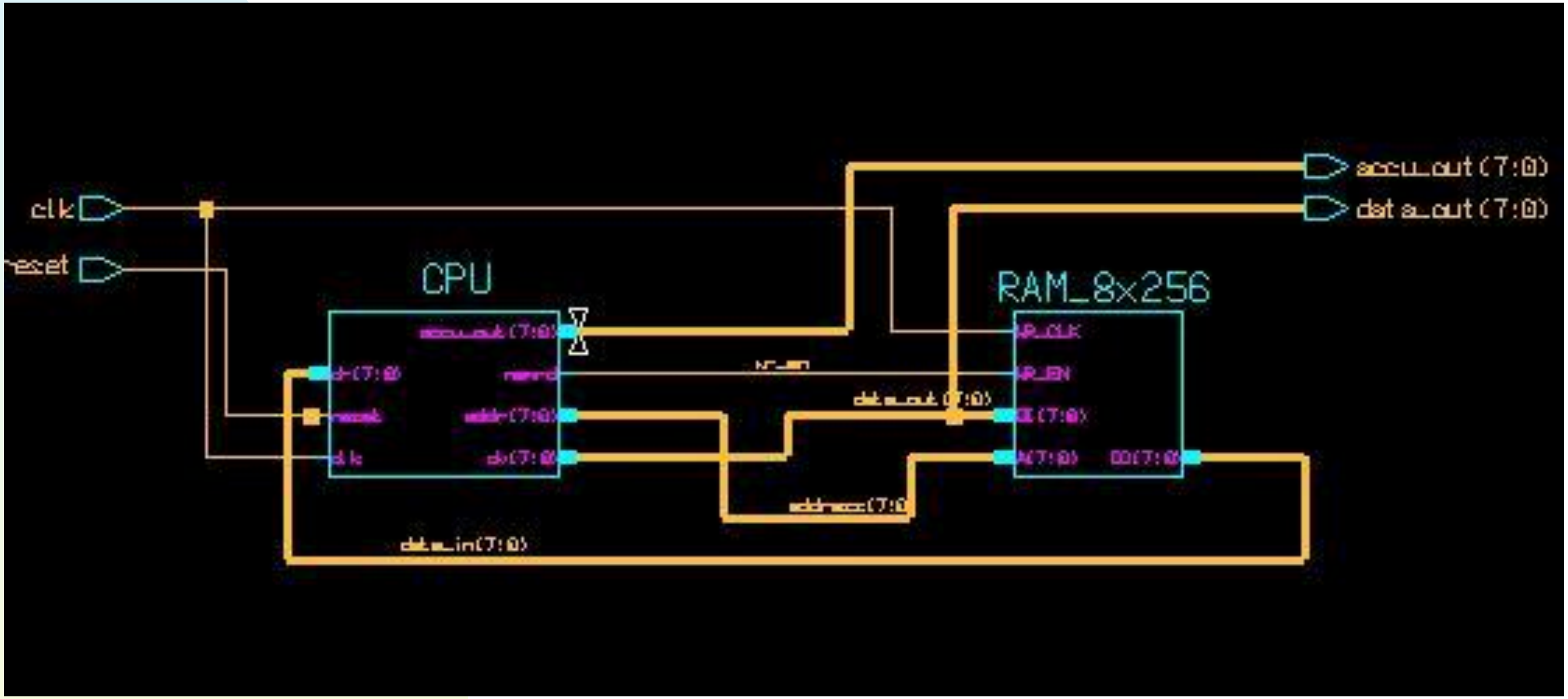


TABLE 12-1: PORTA SFR SUMMARY

Virtual Address	Name		Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
BF88_6000	TRISA	31:24	—	—	—	—	—	—	—	—
		23:16	—	—	—	—	—	—	—	—
		15:8	TRISA15	TRISA14	—	—	—	TRISA10	TRISA9	—
		7:0	TRISA<7:0>							
BF88_6004	TRISACLR	31:0	Write clears selected bits in TRISA, read yields undefined value							
BF88_6008	TRISASET	31:0	Write sets selected bits in TRISA, read yields undefined value							
BF88_600C	TRISAINV	31:0	Write inverts selected bits in TRISA, read yields undefined value							
BF88_6010	PORTA	31:24	—	—	—	—	—	—	—	—
		23:16	—	—	—	—	—	—	—	—
		15:8	RA15	RA14	—	—	—	RA10	RA9	—
		7:0	RA<7:0>							
BF88_6014	PORTACLR	31:0	Write clears selected bits in PORTA, read yields undefined value							
BF88_6018	PORTASET	31:0	Write sets selected bits in PORTA, read yields undefined value							
BF88_601C	PORTAINV	31:0	Write inverts selected bits in PORTA, read yields undefined value							
BF88_6020	LATA	31:24	—	—	—	—	—	—	—	—
		23:16	—	—	—	—	—	—	—	—
		15:8	LATA15	LATA14	—	—	—	LATA10	LATA9	—
		7:0	LATA<7:0>							
BF88_6024	LATACLR	31:0	Write clears selected bits in LATA, read yields undefined value							
BF88_6028	LATASET	31:0	Write sets selected bits in LATA, read yields undefined value							
BF88_602C	LATAINV	31:0	Write inverts selected bits in LATA, read yields undefined value							
BF88_6030	ODCA	31:24	—	—	—	—	—	—	—	—
		23:16	—	—	—	—	—	—	—	—
		15:8	ODCA15	ODCA14	—	—	—	ODCA10	ODCA9	—
		7:0	ODCA<7:0>							
BF88_6034	ODCACLR	31:0	Write clears selected bits in ODCA, read yields undefined value							
BF88_6038	ODCFASET	31:0	Write sets selected bits in ODCA, read yields undefined value							
BF88_603C	ODCAINV	31:0	Write inverts selected bits in ODCA, read yields undefined value							

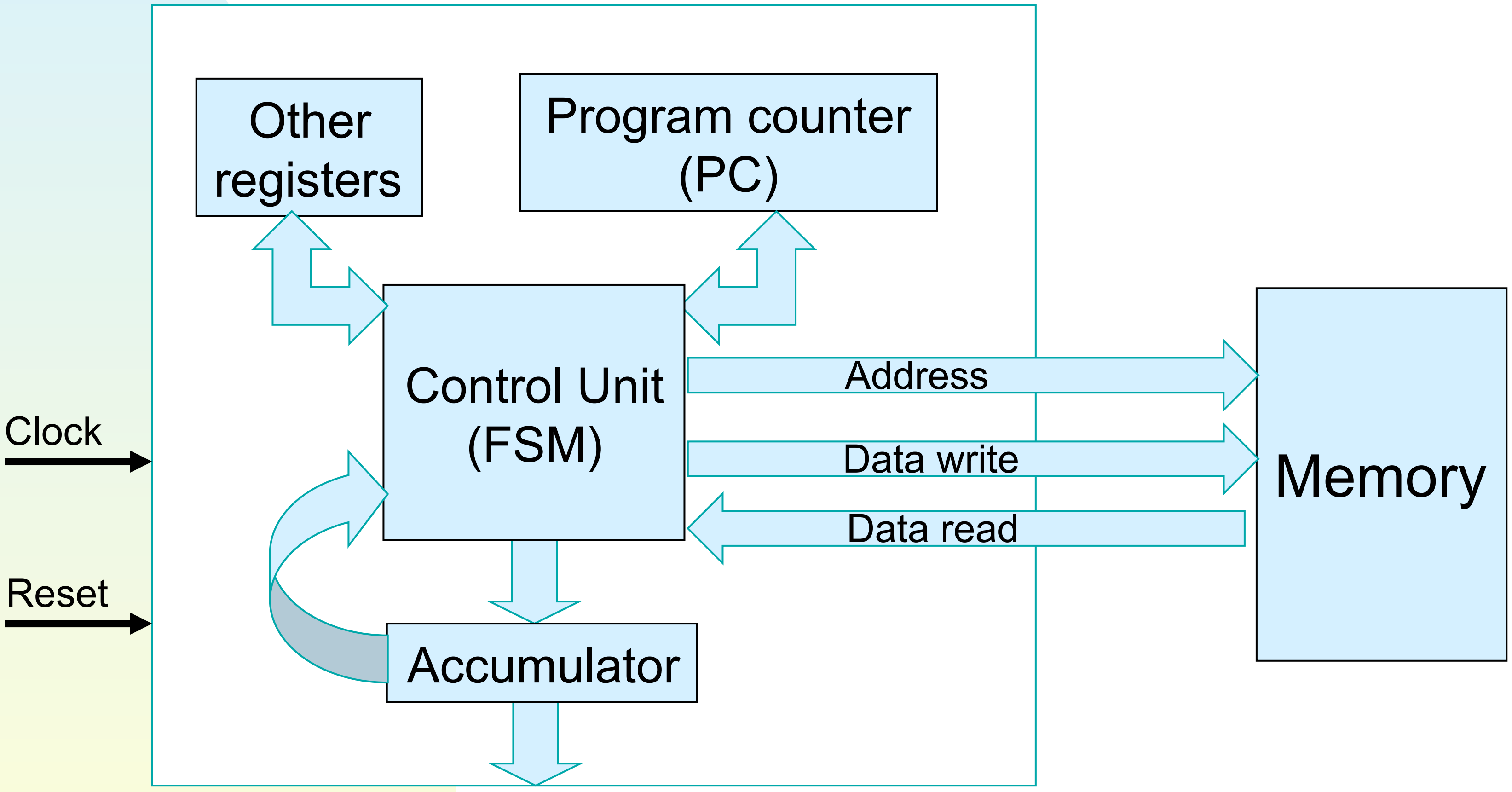
Lab 6: design a microprocessor



Lab 6 overview:

- Start with "skeleton" code for a simple CPU
  - available for download from course home page
- Build up functionality so that the CPU can run a sample program containing five opcodes
- Expand the design on your own…
  - New opcodes to run more complex programs.
  - Peripherals
  - Etc.

Lab 6 CPU (von Neumann):



Your first program

Address	Data	Comment
00	01	code for **LDA**
01	07	value 7
02	03	code for **ADD**
03	0A	address 0A
04	02	code for **STA**
05	10	address 10
06	04	code for **JNC**
07	02	address 02
08	05	code for **JMP**
09	00	address 00
0A	09	Value 09 stored at address 0A

- Five opcodes: LDA, ADD, STA, JNC, JMP
- Some opcodes take more than one CPU cycle, extra registers

How to start Lab 6

- Go to the online write-up on the course page
- From there, you will find links to:
  - Entity: cpu.vhdl
  - Architecture: cpu-fsm.vhdl
  - Memory: procram.vhdl
  - 7 segment display: disp4.vhd
- Download these files and use them as a starting point
- Discuss ideas for further development with the instructor