Architectures for Programmable DSPs

Dr. Deepa Kundur

University of Toronto

Basic Architectural Features

- A digital signal processor is a specialized microprocessor for the purpose of real-time DSP computing.
- DSP applications commonly share the following characteristics:
  - Algorithms are mathematically intensive; common algorithms require many multiply and accumulates.
  - Algorithms must run in real-time; processing of a data block must occur before next block arrives.
  - Algorithms are under constant development; DSP systems should be flexible to support changes and improvements in the state-of-the-art.

Basic Architectural Features

- Programmable DSPs should provide for instructions that one would find in most general microprocessors.
- The basic instruction capabilities (provided with dedicated high-speed hardware) should include:
  - arithmetic operations: add, subtract and multiply
  - logic operations: `AND`, `OR`, `XOR`, and `NOT`
  - multiply and accumulate (MAC) operation
  - signal scaling operations before and/or after digital signal processing

Basic Architectural Features

- Support architecture should include:
  - RAM; i.e., on-chip memories for signal samples
  - ROM; on-chip program memory for programs and algorithm parameters such as filter coefficients
  - on-chip registers for storage of intermediate results

DSP Computational Building Blocks

- Multiplier
- Shifter
- Multiply and accumulate (MAC) unit
- Arithmetic logic unit

Multiplier

- The following specifications are important when designing a multiplier:
  - speed $\longrightarrow$ decided by architecture which trades off with circuit complexity and power dissipation
  - accuracy $\longrightarrow$ decided by format representations (number of bits and fixed/floating pt)
  - dynamic range $\longrightarrow$ decided by format representations

Parallel Multiplier

- Advances in speed and size in VLSI technology have made hardware implementation of parallel or array multipliers possible.
- Parallel multipliers implement a complete multiplication of two binary numbers to generate the product within a single processor cycle!

Parallel Multiplier: Bit Expansion

Consider the multiplication of two unsigned fixed-point integer numbers A and B where A is m-bits $(A_{m-1}, A_{m-2}, \ldots, A_0)$ and B is n-bits $(B_{n-1}, B_{n-2}, \ldots, B_0)$:

$$A = \sum_{i=0}^{m-1} A_i 2^i; \quad 0 \leq A \leq 2^m - 1, A_i \in \{0, 1\}$$

$$B = \sum_{j=0}^{n-1} B_j 2^j; \quad 0 \leq B \leq 2^n - 1, B_i \in \{0, 1\}$$

Generally, we will require r-bits where $r > \max(m, n)$ to represent the product $P = A \cdot B$; known as bit expansion.

Parallel Multiplier: Bit Expansion

Q: How many bits are required to represent $P = A \cdot B$?

- Let the minimum number of bits needed to represent the range of P be given by r.
- An r-bit unsigned fixed-point integer number can represent values between 0 and $2^r - 1$.
- Therefore, $0 \leq P \leq 2^r - 1$.

$$\begin{aligned} P_{min} &= A_{min} \cdot B_{min} = 0 \cdot 0 = 0 \\ P_{max} &= A_{max} \cdot B_{max} = (2^m - 1) \cdot (2^n - 1) \\ &= 2^{n+m} - 2^m - 2^n + 1 \end{aligned}$$

Parallel Multiplier: Bit Expansion

Rephrased Q: How many bits are required to represent P_{max}?

$$\begin{aligned} P_{max} &= 2^{n+m} \underbrace{-2^m - 2^n + 1}_{<-1} < 2^{n+m} - 1 \quad \text{for positive } n, m. \\ P_{max} &= 2^{n+m} - 2^m - 2^n + 1 \approx 2^{n+m} \quad \text{for large } n, m. \end{aligned}$$

Therefore, $P_{max} < 2^{n+m}$ is a tight bound.

Parallel Multiplier: Bit Expansion

Rephrased Q: How many bits are required to represent P_{max}? Therefore,

$$r = \lceil \log_2 (P_{max}) \rceil = \log_2 \left(2^{n+m}\right) = m + n$$

for large n, m.

Parallel Multiplier: $n = m = 4$

Example: $m = n = 4$; note: $r = m + n = 4 + 4 = 8$.

$$\begin{aligned} P &= A \cdot B = \sum_{i=0}^{4-1} A_i 2^i \cdot \sum_{i=0}^{4-1} B_i 2^i \\ &= (A_0 2^0 + A_1 2^1 + A_2 2^2 + A_3 2^3) \cdot (B_0 2^0 + B_1 2^1 + B_2 2^2 + B_3 2^3) \\ &= A_0 B_0 2^0 + (A_0 B_1 + A_1 B_0) 2^1 + (A_0 B_2 + A_1 B_1 + A_2 B_0) 2^2 \\ &\quad + (A_0 B_3 + A_1 B_2 + A_2 B_1 + A_3 B_0) 2^3 + (A_1 B_3 + A_2 B_2 + A_3 B_1) 2^4 \\ &\quad + (A_2 B_3 + A_3 B_2) 2^5 + (A_3 B_3) 2^6 \\ &= P_0 2^0 + P_1 2^1 + P_2 2^2 + P_3 2^3 + P_4 2^4 + P_5 2^5 + P_6 2^6 + \underbrace{P_7 2^7}_{???} \end{aligned}$$

Parallel Multiplier: $n = m = 4$

Recall base 10 addition.
Example: (3785 + 6584)

CARRY-OVER	1	1	0	0	
	3	7	8	5	
+	6	5	8	4	
	1	0	3	6	9

↑
CARRY-OVER

Parallel Multiplier: $n = m = 4$

Example: $m = n = 4$; note: $r = m + n = 4 + 4 = 8$.

$$\begin{aligned} P &= A \cdot B = \sum_{i=0}^{4-1} A_i 2^i \cdot \sum_{i=0}^{4-1} B_i 2^i \\ &= (A_0 2^0 + A_1 2^1 + A_2 2^2 + A_3 2^3) \cdot (B_0 2^0 + B_1 2^1 + B_2 2^2 + B_3 2^3) \\ &= A_0 B_0 2^0 + (A_0 B_1 + A_1 B_0) 2^1 + (A_0 B_2 + A_1 B_1 + A_2 B_0) 2^2 \\ &\quad + (A_0 B_3 + A_1 B_2 + A_2 B_1 + A_3 B_0) 2^3 + (A_1 B_3 + A_2 B_2 + A_3 B_1) 2^4 \\ &\quad + (A_2 B_3 + A_3 B_2) 2^5 + (A_3 B_3) 2^6 \\ &= P_0 2^0 + P_1 2^1 + P_2 2^2 + P_3 2^3 + P_4 2^4 + P_5 2^5 + P_6 2^6 + P_7 2^7 \end{aligned}$$

Need to compensate for carry-over bits!

Parallel Multiplier: $n = m = 4$

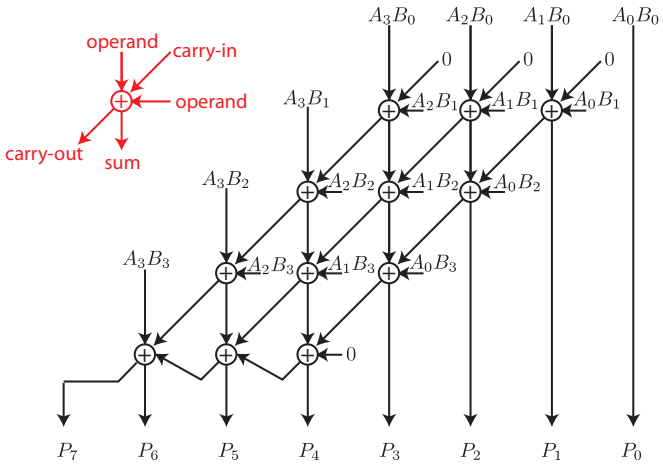
Need to compensate for carry-over bits!

$$\begin{aligned} P_0 &= A_0 B_0 \\ P_1 &= A_0 B_1 + A_1 B_0 + 0 \\ P_2 &= A_0 B_2 + A_1 B_1 + A_2 B_0 + \text{PREV CARRY OVER} \\ &\vdots \\ P_6 &= A_3 B_3 + \text{PREV CARRY OVER} \\ P_7 &= \text{PREV CARRY OVER} \end{aligned}$$

Parallel Multiplier: Braun Multiplier

Example:

- structure of a 4×4 Braun multiplier; i.e., $m = n = 4$



Parallel Multiplier: Braun Multiplier

- Speed: for parallel multiplier the multiplication time is only the longest path delay time through the gates and adders (well within one processor cycle)
- Note: additional hardware before and after the Braun multiplier is required to deal with signed numbers represented in two's complement form.

Parallel Multiplier

- Bus Widths: straightforward implementation requires two buses of width n-bits and a third bus of width $2n$-bits, which is expensive to implement

- To avoid complex bus implementations:
  - program bus can be reused after the multiplication instruction is fetched
  - bus for X can be used for Z by discarding the lower n bits of Z or by saving Z at two successive memory locations

Shifter

- required to scale down or scale up operands and results to avoid errors resulting from overflows and underflows during computations
- Note: When computing the sum of N numbers, each represented by n-bits, the overall sum will have $n + \log_2 N$ bits
- **Q:** Why?
  - Each number is represented with n bits.
  - For the sum of N numbers, $P_{max} = N \times (2^n - 1)$.
  - Therefore,
    $r = \log_2 P_{max} \approx \log_2 (N \times 2^n) = \log_2 2^n + \log_2 N = n + \log_2 N$.

Shifter

- **Q:** When is scaling useful?
  - to avoid overflow can scale down each of the N numbers by $\log_2 N$ bits before conducting the sum
  - to obtain actual sum scale up the result by $\log_2 N$ bits when required
  - trade-off between overflow prevention and accuracy

Shifter

- Example: Suppose $n = 4$ and we are summing $N = 3$ unsigned fixed point integers as follows:

$$\begin{aligned} S &= x_1 + x_2 + x_3 \\ x_1 &= 10 = [1\ 0\ 1\ 0] \\ x_2 &= 5 = [0\ 1\ 0\ 1] \\ x_3 &= 8 = [1\ 0\ 0\ 0] \\ S &= 10 + 5 + 8 = 23 > 2^4 - 1 = 15 \end{aligned}$$

- Must scale numbers down by at least $\log_2 N = \log_2 3 \approx 1.584 < 2$.
- Require scaling through a single right-shift.

Shifter

- To scale numbers down by a factor of 2:

$$\begin{aligned} x_1 &= 10 = [1\ 0\ 1\ 0] & \hat{x}_1 &= [0\ 1\ 0\ 1] = 5 \\ x_2 &= 5 = [0\ 1\ 0\ 1] & \hat{x}_2 &= [0\ 0\ 1\ 0] = 2 \neq \tfrac{5}{2} \\ x_3 &= 8 = [1\ 0\ 0\ 0] & \hat{x}_3 &= [0\ 1\ 0\ 0] = 4 \end{aligned}$$

- To add:

$$\hat{S} = \hat{x}_1 + \hat{x}_2 + \hat{x}_3 = 5 + 2 + 4 = 11 = [1\ 0\ 1\ 1]$$

- To scale sum up by a factor of 2 (allow bit expansion here):

$$\tilde{S} = [1\ 0\ 1\ 1\ 0] = 22 \neq 23 = 10 + 5 + 8 = S$$

Shifter

- Consider the following related example. To scale numbers down by a factor of 2:

$$\begin{aligned} x_1 &= 11 = [1\ 0\ 1\ 1] & \hat{x}_1 &= [0\ 1\ 0\ 1] = 5 \neq \tfrac{11}{2} \\ x_2 &= 5 = [0\ 1\ 0\ 1] & \hat{x}_2 &= [0\ 0\ 1\ 0] = 2 \neq \tfrac{5}{2} \\ x_3 &= 9 = [1\ 0\ 0\ 1] & \hat{x}_3 &= [0\ 1\ 0\ 0] = 4 \neq \tfrac{9}{2} \end{aligned}$$

- To add:

$$\hat{S} = \hat{x}_1 + \hat{x}_2 + \hat{x}_3 = 5 + 2 + 4 = 11 = [1\ 0\ 1\ 1]$$

- To scale sum up by a factor of 2 (allow bit expansion here):

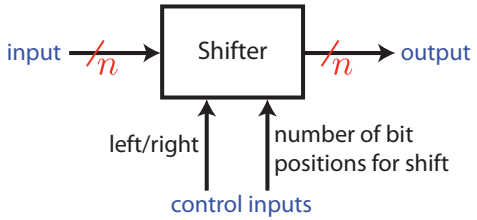
$$\tilde{S} = [1\ 0\ 1\ 1\ 0] = 22 \neq 25 = 11 + 5 + 9 = S$$

Shifter

- **Q:** When is scaling useful?
  - Conducting floating point additions, where each operand should be normalized to the same exponent prior to addition
  - one of the operands can be shifted to the required number of bit positions to equalize the exponents

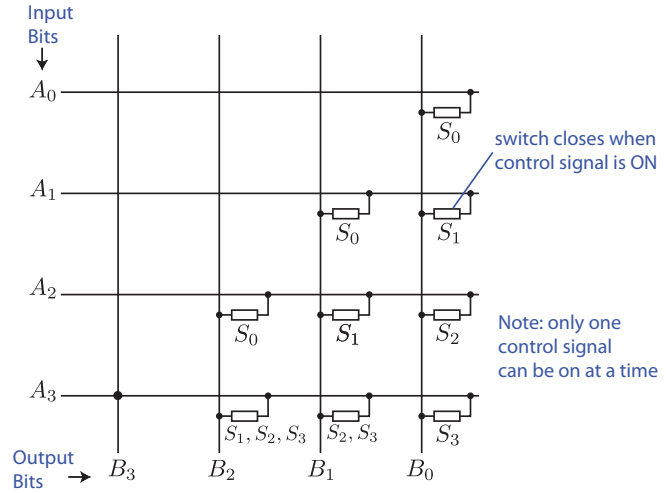
Barrel Shifter

- Shifting in conventional microprocessors is implemented by an operation similar to one in a shift register taking one clock cycle for every single bit shift.
  - Many shifts are often required creating a latency of multiple clock cycles.
- Barrel shifters allow shifting of multiple bit positions within one clock cycle reducing latency for real-time DSP computations.



Barrel Shifter

Implementation of a 4-bit shift-right barrel shifter:



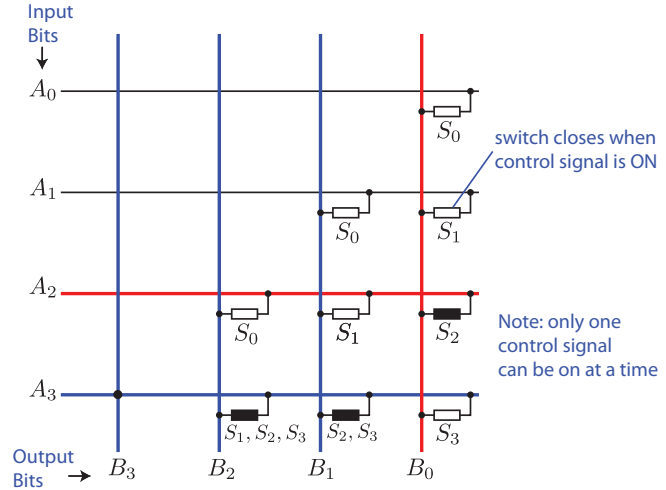
Barrel Shifter

Implementation of a 4-bit shift-right barrel shifter:

Input	Shift (Switch)	Output ($B_3B_2B_1B_0$)
$A_3A_2A_1A_0$	0 (S_0)	$A_3A_2A_1A_0$
$A_3A_2A_1A_0$	1 (S_1)	$A_3A_3A_2A_1$
$A_3A_2A_1A_0$	2 (S_2)	$A_3A_3A_3A_2$
$A_3A_2A_1A_0$	3 (S_3)	$A_3A_3A_3A_3$

- logic circuit takes a fraction of a clock cycle to execute
- majority of delay is in decoding the control lines and setting up the path from the input lines to the output lines

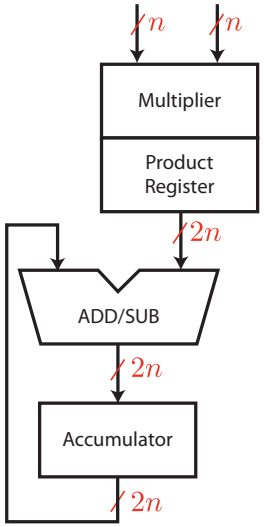
Barrel Shifter



Multiply and Accumulate

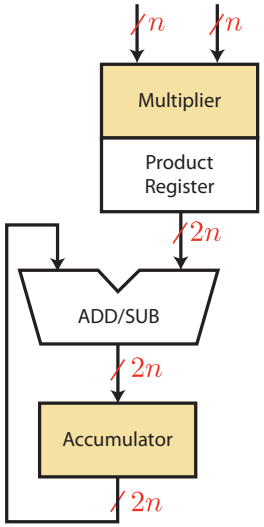
- multiply and accumulate (MAC) unit performs the accumulation of a series of successively generated products
- common operation in DSP applications such as filtering

MAC Unit Configuration



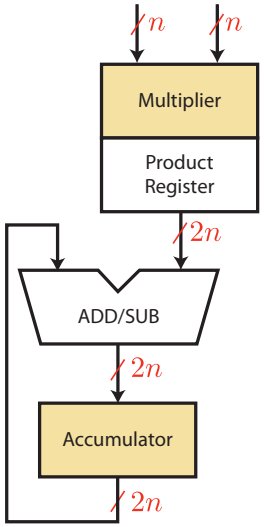
- can implement $A + BC$ operations
- clearing the accumulator at the right time (e.g., as an initialization to zero) provides appropriate sum of products

MAC Unit Configuration



- multiplication and accumulation each require a separate instruction execution cycle
- however, they can work in parallel
  - when multiplier is working on current product, the accumulator works on adding previous product

MAC Unit Configuration



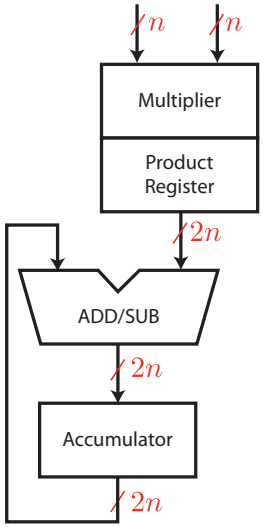
- if N products are to be accumulated, $N - 1$ multiplies can overlap with the accumulation
  - during the first multiply, the accumulator is idle
  - during the last accumulate, the multiplier is idle since all N products have been computed
- to compute a MAC for N products, $N + 1$ instruction execution cycles are required
- for $N \gg 1$, works out to almost one MAC operation per instruction cycle

MAC Unit

Q: If a sum of 256 products is to be computed using a pipelined MAC unit and if the MAC execution time of the unit is 100 ns, what is the total time required to compute the operation?

A:
For 256 MAC operations, need 257 execution cycles.
Total time required $= 257 \times 100 \times 10^{-9}$ sec $= \underline{25.7}\mu$s

MAC Unit Overflow and Underflow



- Strategies to address overflow or underflow:
  - accumulator guard bits (i.e., extra bits for the accumulator) added; implication: size of ADD/SUB unit will increase
  - barrel shifters at the input and output of MAC unit needed to normalize values
  - saturation logic used to assign largest (smallest) values to accumulator when overflow (underflow) occurs

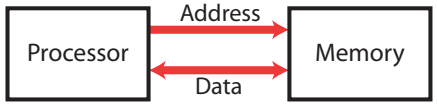
Arithmetic and Logic Unit

- arithmetic logic unit (ALU) carries out additional arithmetic and logic operations required for a DSP:
  - add, subtract, increment, decrement, negate
  - `AND, OR, NOT, XOR`, compare
  - shift, multiply (uncommon to general microprocessors)
- with additional features common to general microprocessors:
  - status flags for sign, zero, carry and overflow
  - overflow management via saturation logic
  - register files for storing intermediate results

Bus Architecture and Memory

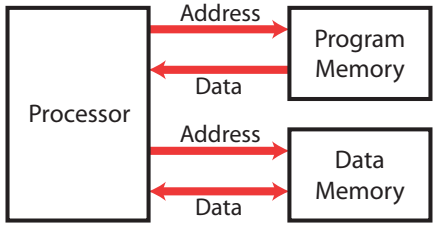
- Bus architecture and memory play a significant role in dictating cost, speed and size of DSPs.
- Common architectures include the von Neumann and Harvard architectures.

von Neumann Architecture



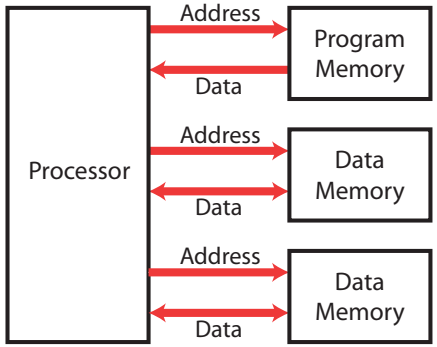
- program and data reside in same memory
- single bus is used to access both
- Implications:
  - slows down program execution since processor has to wait for data even after instruction is made available

Harvard Architecture



- program and data reside in separate memories with two independent buses
- Implications:
  - faster program execution because of simultaneous memory access capability
- What if there are two operands?

Another Possible DSP Bus Structure



- suited for operations with two operands (e.g., multiplication)
- Implications: requires hardware and interconnections increasing cost
  hardware complexity-speed trade-off needed!

On-Chip Memory

- on-chip = on-processor
- help in running the DSP algorithms faster than when memory is off-chip
  - dedicated addresses and data buses are available
- speed: on-chip memories should match the speeds of the ALU operations
- size: the more area chip memory takes, the less area available for other DSP functions

On-Chip Memory

- Wish List:
  - all memory should reside on-chip!
  - separate on-chip program and data spaces
  - on-chip data space partitioned further into areas for data samples, coefficients and results
- Implication: area dedicated to memory will be so large that basic DSP functions may not be implementable on-chip!

Practical Organization of On-Chip Memory

- for DSP algorithms requiring repeated executions of a single instruction (e.g., MAC):
  - instructions can be placed in external memory and once fetched can be placed in the instruction cache
  - the result is normally saved at the end, so external memory can be employed
  - only two data memories for operands can be placed on-chip

Practical Organization of On-Chip Memory

- using dual-access on-chip memories (can be accessed twice per instruction cycle):
  - can get away with only two on-chip memories for instructions such as multiply
  - instruction fetch + two operand fetches + memory access to save result can all be done in one clock cycle
  - can configure on-chip memory for different uses at different times

Data Addressing Capabilities

- efficient way of accessing data (signal sample and filter coefficients) can significantly improve implementation performance
- flexible ways to access data helps in writing efficient programs
- data addressing modes enhance DSP implementations

DSP Addressing Modes

- immediate
- register
- direct
- indirect
- special addressing modes:
  - circular
  - bit-reversed

Immediate Addressing Mode

- operand is explicitly known in value
- capability to include data as part of the instruction

Instruction	**Operation**
$ADD\ \#imm$	$\#imm + A \rightarrow A$

- $\#imm$: value represented by imm (fixed number such as filter coefficient is known ahead of time)
- A: accumulator register

Register Addressing Mode

- operand is always in processor register reg
- capability to reference data through its register

Instruction	**Operation**
$ADD\ reg$	$reg + A \rightarrow A$

- reg: processor register provides operand
- A: accumulator register

Direct Addressing Mode

- operand is always in memory location mem
- capability to reference data by giving its memory location directly

Instruction	**Operation**
$ADD\ mem$	$mem + A \rightarrow A$

- mem: specified memory location provides operand (e.g., memory could hold input signal value)
- A: accumulator register

Indirect Addressing Mode

- operand memory location is variable
- operand address is given by the value of register *addrreg*
- operand accessed using pointer *addrreg*

Instruction	Operation
$ADD *addrreg$	$*addrreg + A \rightarrow A$

- *addrreg*: needs to be loaded with the register location before use
- *A*: accumulator register

Special Addressing Modes

- Circular Addressing Mode: circular buffer allows one to handle a continuous stream of incoming data samples; once the end of the buffer is reached, samples are wrapped around and added to the beginning again
  - useful for implementing real-time digital signal processing where the input stream is effectively continuous
- Bit-Reversed Addressing Mode: address generation unit can be provided with the capability of providing bit-reversed indices
  - useful for implementing radix-2 FFT (fast Fourier Transform) algorithms where either the input or output is in bit-reversed order

Special Addressing Modes

Circular Addressing:

- Can avoid constantly testing for the need to wrap.
- Suppose we consider eight registers to store an incoming data stream.

Reference Index	Address
$0 = 0 \bmod 8 = 8 \bmod 8 = 16 \bmod 8 \cdots$	$000 = 0$
$1 = 1 \bmod 8 = 9 \bmod 8 = 17 \bmod 8 \cdots$	$001 = 1$
$2 = 2 \bmod 8 = 10 \bmod 8 = 18 \bmod 8 \cdots$	$010 = 2$
$3 = 3 \bmod 8 = 11 \bmod 8 = 19 \bmod 8 \cdots$	$011 = 3$
$4 = 4 \bmod 8 = 12 \bmod 8 = 20 \bmod 8 \cdots$	$100 = 4$
$5 = 5 \bmod 8 = 13 \bmod 8 = 21 \bmod 8 \cdots$	$101 = 5$
$6 = 6 \bmod 8 = 14 \bmod 8 = 22 \bmod 8 \cdots$	$110 = 6$
$7 = 7 \bmod 8 = 15 \bmod 8 = 23 \bmod 8 \cdots$	$111 = 7$

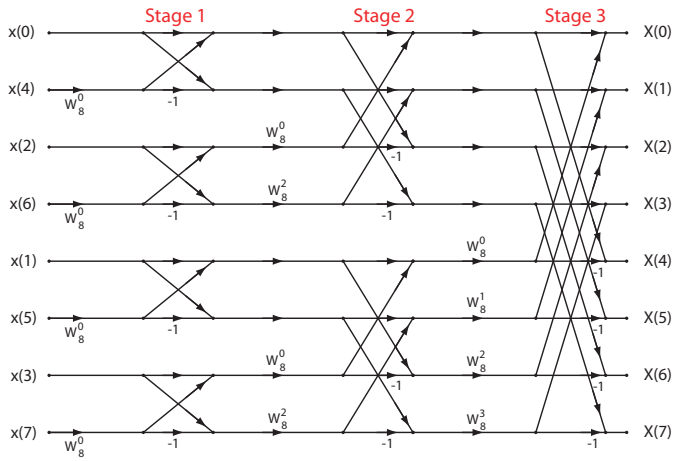
Special Addressing Modes

Bit-Reversed Addressing:

Input Index	Output Index
000 = 0	000 = 0
001 = 1	100 = 4
010 = 2	010 = 2
011 = 3	110 = 6
100 = 4	001 = 1
101 = 5	101 = 5
110 = 6	011 = 3
111 = 7	111 = 7

Special Addressing Modes

Bit-Reversed Addressing: Why?



Speed Issues

- fast execution of algorithms is the most important requirement of a DSP architecture
  - high speed instruction operation
  - large throughputs
- facilitated by advances in VLSI technology and design innovations

Hardware Architecture

- dedicated hardware support for multiplications, scaling, loops and repeats, and special addressing modes are essential for fast DSP implementations
- Harvard architecture significantly improves program execution time compared to von Neumann
- on-chip memories aid speed of program execution considerably

Parallelism

Parallelism means:

- provision of multiple function units, which may operate in parallel to increase throughput
  - multiple memories
  - different ALUs for data and address computations
- advantage: algorithms can perform more than one operation at a time increasing speed
- disadvantage: complex hardware required to control units and make sure instructions and data can be fetched simultaneously

Pipelining

- architectural feature in which an instruction is broken into a number of steps
  - a separate unit performs each step at the same time usually working on different stage of data
- advantage: if repeated use of the instruction is required, then after an initial latency the output throughput becomes one instruction per unit time
- disadvantages: pipeline latency, having to break instructions up into equally-timed units

Pipelining Example

Five steps:
Step 1: `instruction fetch`
Step 2: `instruction decode`
Step 3: `operand fetch`
Step 4: `execute`
Step 5: `save`

Time Slot	Step 1	Step 2	Step 3	Step 4	Step 5	Result
t_0	Inst 1					
t_1	Inst 2	Inst 1				
t_2	Inst 3	Inst 2	Inst 1			
t_3	Inst 4	Inst 3	Inst 2	Inst 1		
t_4	Inst 5	Inst 4	Inst 3	Inst 2	Inst 1 complete	
t_5	Inst 6	Inst 5	Inst 4	Inst 3	Inst 2 complete	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Simplifying assumption: all steps take equal time

System Level Parallelism and Pipelining

Consider 8-tap FIR filter:

$$\begin{aligned} y(n) &= \sum_{k=0}^{7} h(k)x(n-k) \\ &= h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + \cdots \\ &\qquad \cdots + h(6)x(n-6) + h(7)x(n-7) \end{aligned}$$

- can be implemented in many ways depending on number of multipliers and accumulators available

System Level Parallelism and Pipelining

Consider 8-tap FIR filter:

- input needed in registers is
$$[x(n)\ x(n-1)\ x(n-2)\ \cdots x(n-7)]$$
- time to produce $y(n)$ = time to process the input block
$$[x(n)\ x(n-1)\ x(n-2)\ \cdots x(n-7)]$$

$$y(n) = h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + \cdots + h(6)x(n-6) + h(7)x(n-7)$$

- new input $x(n+1)$ can be processed after $y(n)$ is produced
- corresponding input needed in registers is
$$[x(n+1)\ x(n)\ x(n-1)\ \cdots x(n-6)]$$

System Level Parallelism and Pipelining

Consider 8-tap FIR filter:

- If it takes T_B time units to process the register block, then for a continuous input stream the throughput is one output sample per T_B time units.
- A new input sample is placed into the register block every T_B time units.

Time	Register Block
0	$[x(n)\ x(n-1)\ x(n-2)\ \cdots x(n-7)]$
T_B	$[x(n+1)\ x(n)\ x(n-1)\ \cdots x(n-6)]$
$2T_B$	$[x(n+2)\ x(n+1)\ x(n)\ \cdots x(n-5)]$
$3T_B$	$[x(n+3)\ x(n+2)\ x(n+1)\ \cdots x(n-4)]$
$\vdots$	$\vdots$

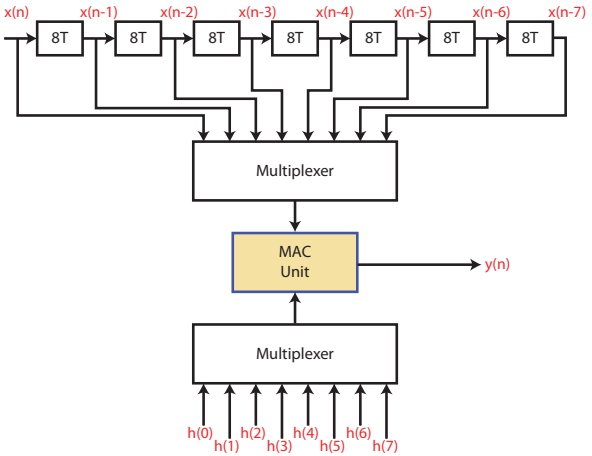
- A shift in the register block every T_B time units is needed to accommodate a new input sample.

System Level Parallelism and Pipelining

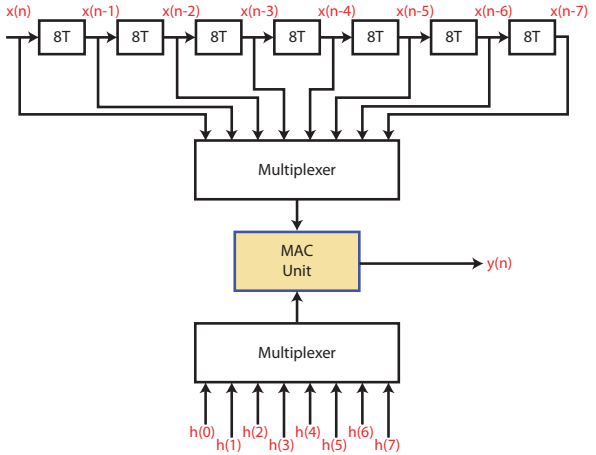
Consider 8-tap FIR filter:

- If the sampling period T_S is larger than T_B, then buffering is needed.
- If the sampling period T_S is less than T_B, then the processor may be idle.
- T_B can be reduced with appropriate parallelism and pipelining.

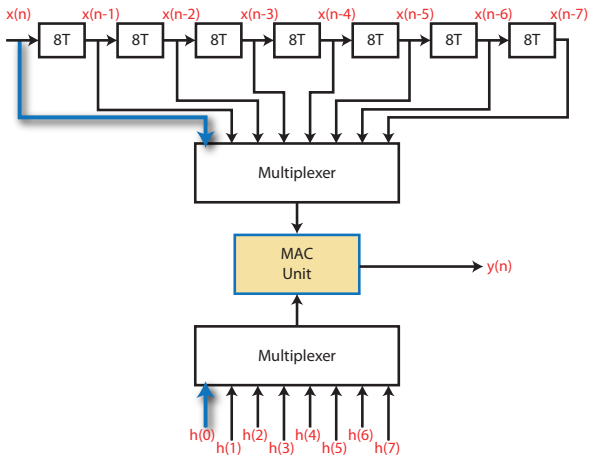
Implementation Using a Single MAC Unit



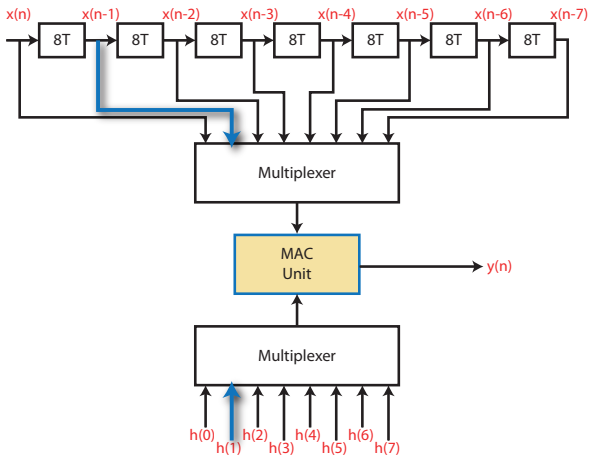
- T: time taken to compute one product term and add it to accumulator
- new input sample can be processed every $8T$ time units; i.e., $T_B = 8T$



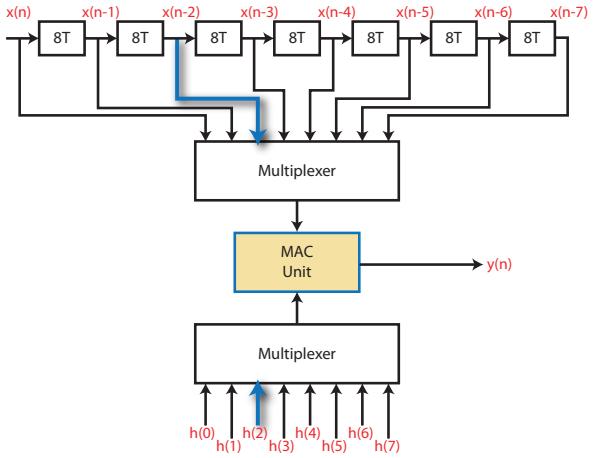
- At $t = 0$, initialization occurs.
- Accumulator $= 0$



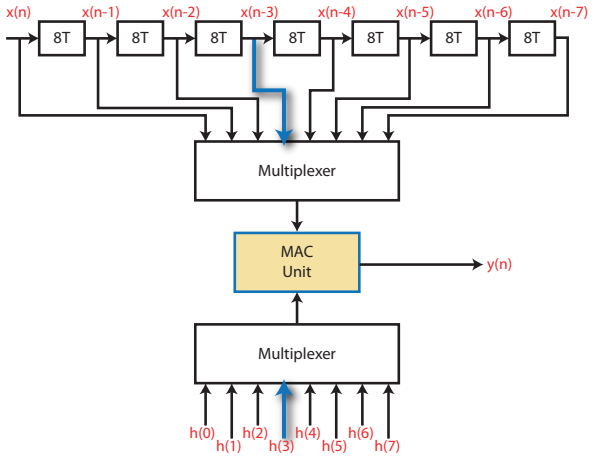
- At $t = T$
- Accumulator $= 0 + h(0)x(n)$



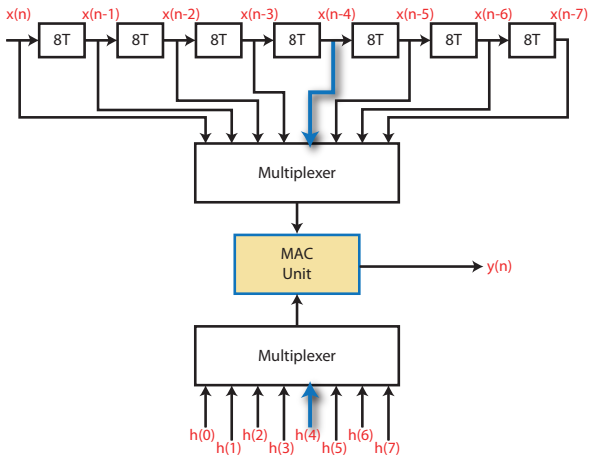
- At $t = 2T$
- Accumulator $= h(0)x(n) + h(1)x(n-1)$



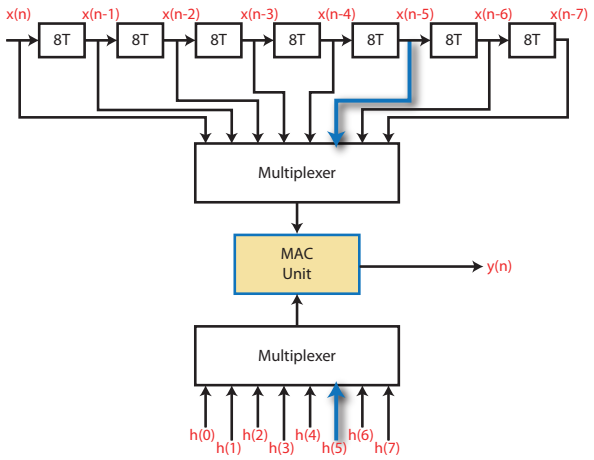
- At $t = 3T$
- Accumulator $= h(0)x(n) + h(1)x(n-1) + h(2)x(n-2)$



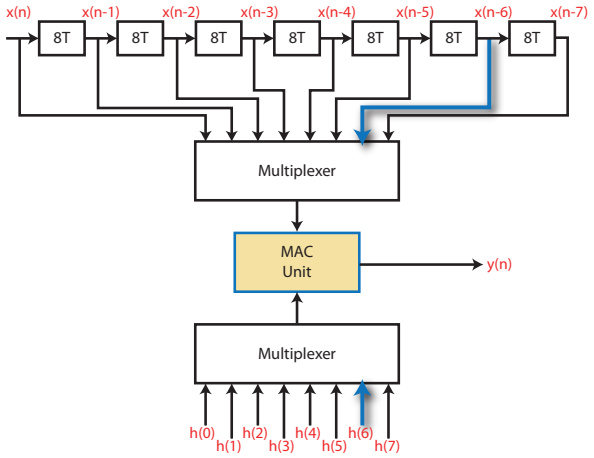
- At $t = 4T$
- Accumulator $= h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + h(3)x(n-3)$



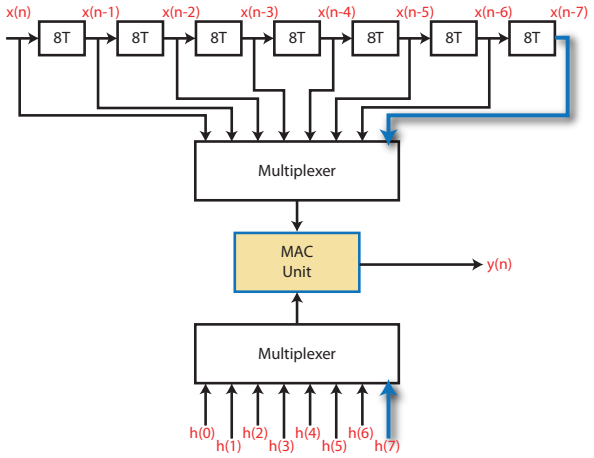
- At $t = 5T$
- Accumulator $= h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + h(3)x(n-3) + h(4)x(n-4)$



- At $t = 6T$
- Accumulator $= h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + h(3)x(n-3) + h(4)x(n-4) + h(5)x(n-5)$

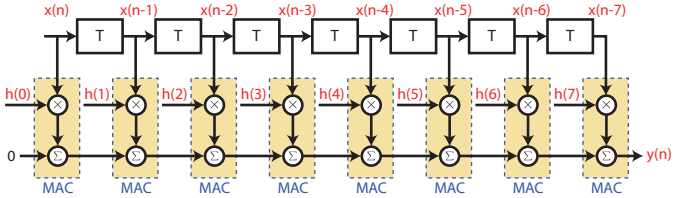


- At $t = 7T$
- Accumulator $= h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + h(3)x(n-3) + h(4)x(n-4) + h(5)x(n-5) + h(6)x(n-6)$



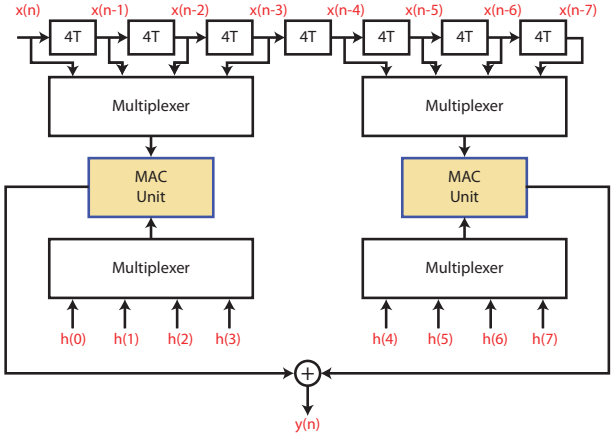
- At $t = 8T$
- Accumulator $= h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + h(3)x(n-3) + h(4)x(n-4) + h(5)x(n-5) + h(6)x(n-6) + h(7)x(n-7)$

Pipelined Implementation: 8 Multipliers and 8 Accumulators



- T: time taken to compute one product term and add it to accumulator
- new input sample can be processed every T time units;
  i.e., $T_B = T$ (8 times faster!)

Parallel Implementation: Two MAC Units



- T: time taken to compute one product term and add it to accumulator
- new input sample can be processed every $4T$ time units;
  i.e., $T_B = 4T$

■