Application Specific Integrated Circuits for Digital Signal Processing
Lecture 8

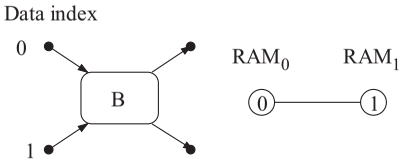
Oscar Gustafsson

Today's topics

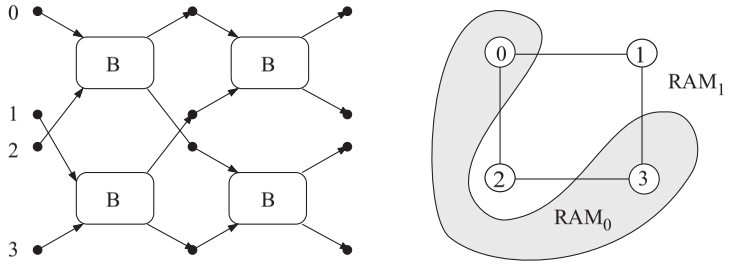
- Resource assignment
- Architectures
- Standard architectures
- Shared memory architectures
- Alternative architectures for easy mapping

FFT case study

- Butterflies have two inputs and two outputs
- Try to keep data in separate memories to avoid multiport memories
- Initially, consider a 2-point FFT and construct exclusion graph

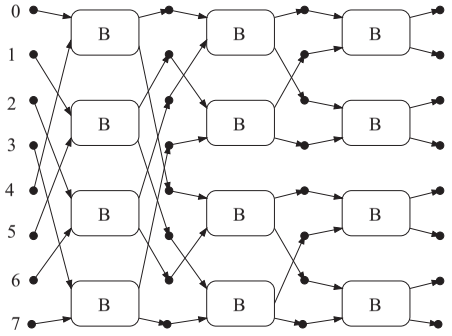


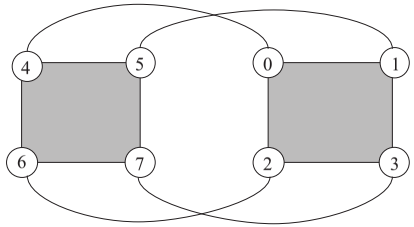
- 4-point FFT



FFT case study

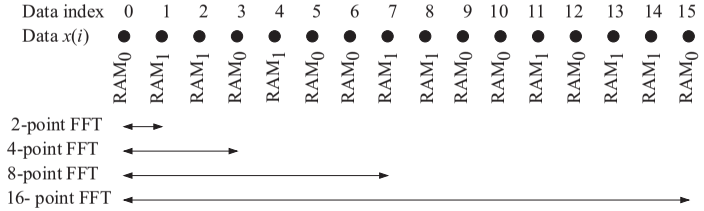
- 8-point FFT





FFT case study

- Generalized solution for 2^n-point FFT with two memories

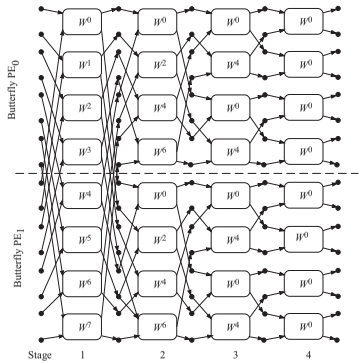


- Assign memory row $r = \sum_{i=0}^{n-1} r_i 2^i$ to $\text{Mem}_M(r)$

$$M(r) = r_0 \oplus r_1 \oplus \cdots \oplus r_{n-1} \tag{1}$$

FFT case study

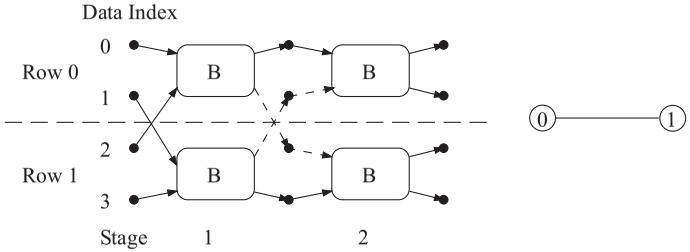
- In an earlier lecture we determined that two butterfly PEs are enough
- How should we map the butterfly operations to the PEs?
- Initial attempt



- Causes much data movement between the two PEs

FFT case study

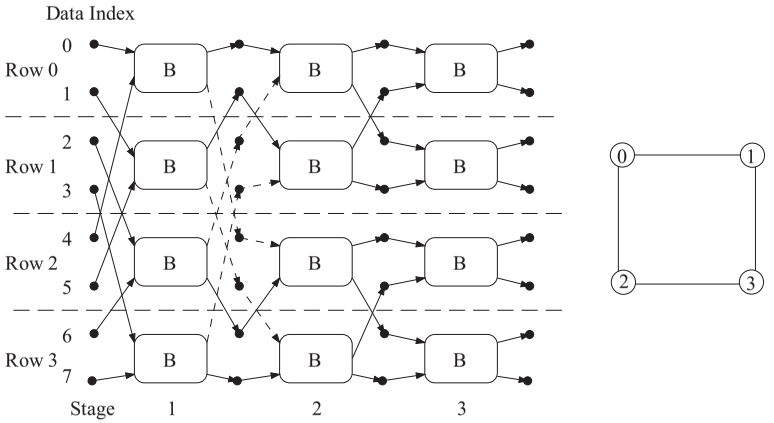
- Want to simplify the interconnect
- Consider 4-point FFT



- Memory rows 1 and 2 are assigned to the same memory (according to the earlier analysis)
- PE rows 0 and 1 mapped the same PE ⇒ Memory storing memory rows 1 and 2 communicates butterfly inputs ⇒ switch required
- Introduce a constraint that this should not happen and obtain the exclusion graph

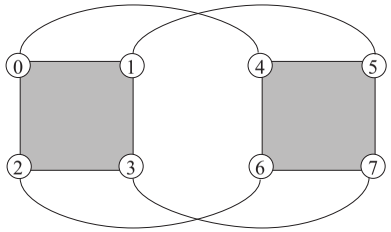
FFT case study

- 8-point FFT



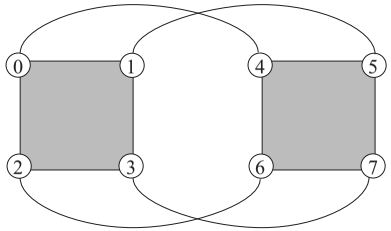
FFT case study

- 16-point FFT exclusion graph



FFT case study

- 16-point FFT exclusion graph

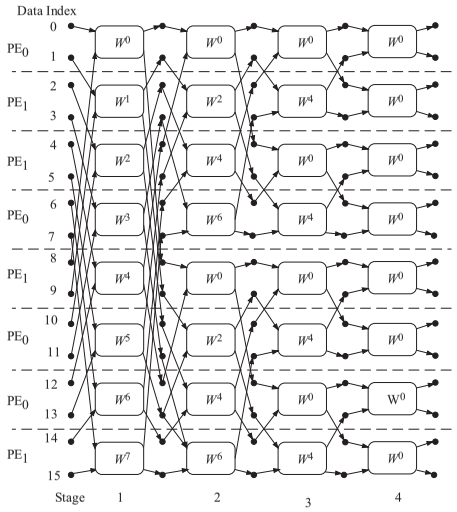


- Generalized solution for 2^n-point FFT with two PEs
- Assign PE row $r = \sum_{i=0}^{n-2} r_i 2^i$ to $\mathrm{PE}_P E(r)$

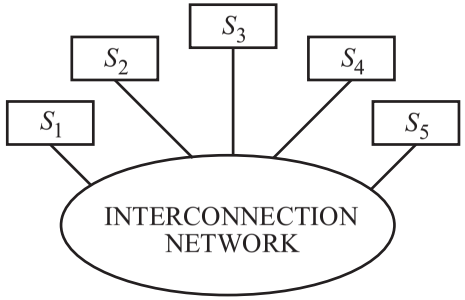
$$PE(r) = r_0 \oplus r_1 \oplus \cdots \oplus r_{n-2} \tag{2}$$

FFT case study

- 16-point FFT result



Architecture

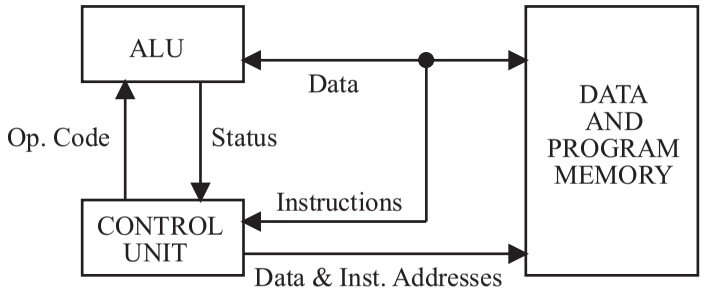


- Several subsystems $S_1, S_2, \ldots$ and interconnect
- Can be seen at different levels:
  - W-CDMA modem (system/application)
  - FFT processor (algorithm)
  - Multiplier (operation)

Typical requirements

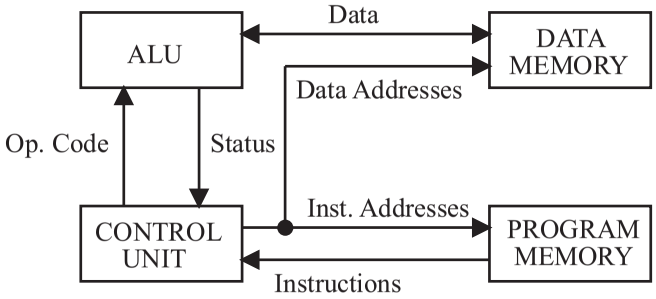
- High processing load
- Parallelism in algorithms
- Use multiple processing elements (PEs) or processors
  - Processor: PE + internal memory + control (can work independently)
- Parallel/distributed architectures

Standard architectures – Von Neumann



- Used in general purpose CPUs
- Shared instruction and data bus/memory
- Limited memory bandwidth for data intensive algorithms

Standard architectures – Harvard



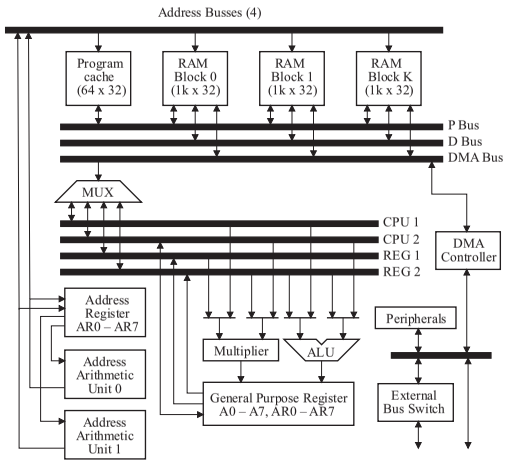
- Separate data and instruction memories/buses
- More suitable for DSP

Standard architectures – DSP processors

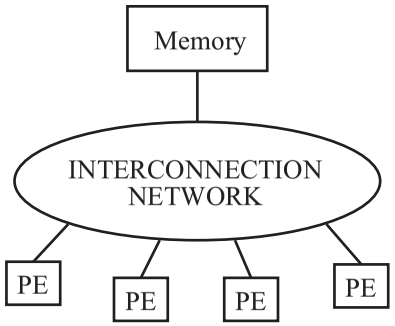
- Common features found in DSP processors
  - Fast multiplier, one cycle MAC ($Z = AB + Z$)
  - Several functional units (ALUs, address generation)
  - On-chip memories (instructions, data, look-up)
  - Often the program can fit in the internal memories
  - Several buses
  - Special addressing (modulo, bit-reversed)
  - Low overhead looping (hardware looping)

Standard architectures – DSP processors

- Example DSP processor

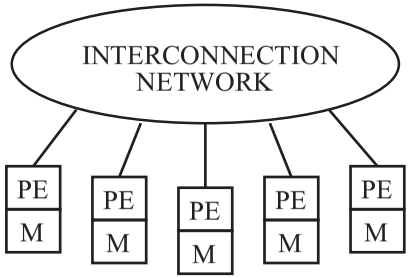


Standard architectures – Multi-processor



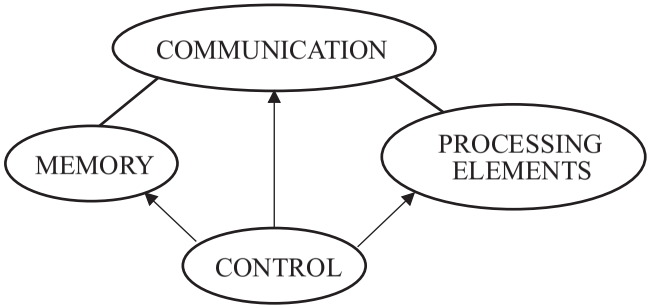
- Shared memory space
  - Tightly coupled: same memory access time
  - Loosely coupled: shorter access time to local memory

Standard architectures – Multi-computer



- Private memory space
- High latency for intercommunication
- Message-based

Generalized architecture



- Need to define these four components of the architecture

Processing elements

- Typical operations:
  - Adder
  - Subtracter
  - Adder/subtracter
  - Adder/subtracter/shifter
  - Multiplier
  - Multiply-accumulate (MAC)
  - Vector product
  - Butterfly
  - Two-port adaptor
- Homogeneous PEs: all PEs can do all work
- Larger operations leads to less communication, but less flexibility in scheduling
- Multiple I/O PEs require multiple memories/ports to fully utilize PE

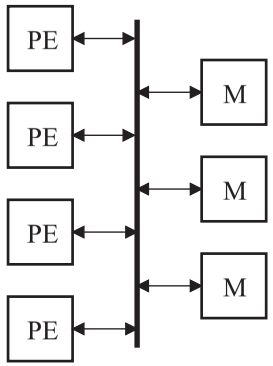
Memory

- Multiple memories
- Multiple ports
  - Multiple read ports somewhat easy
- Main challenge: avoid memory conflicts

Interconnection

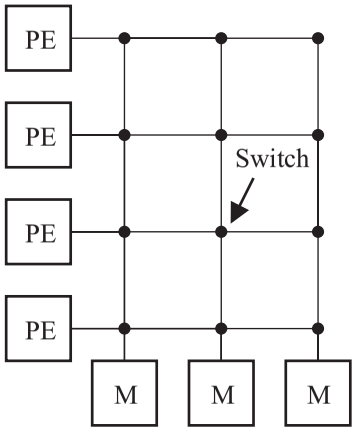
- Two different types of communication
  - Packet switching – send and eventually it will arrive (mail)
  - Circuit switching – establish connection (telephone)
- Many different topologies suggested

Interconnect topologies – Bus



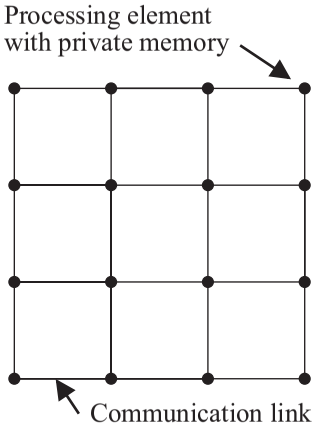
- Simple control
- Every unit can connect to every other
- Possibly performance problems as only two units can communicate at a time

Interconnect topologies – Switch bar



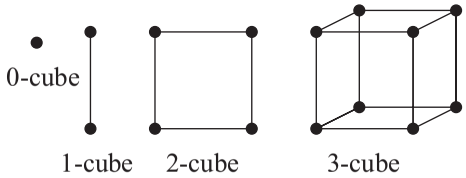
- Non-trivial control
- Every unit can connect to every other
- Possible to establish several communication channels

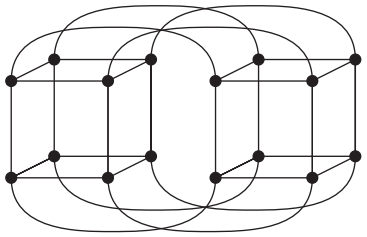
Interconnect topologies – Mesh



- Connect in a regular way
- Units can connect to neighbors
- Possible to establish several communication channels

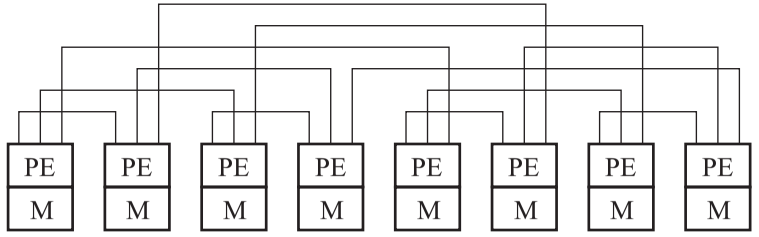
Interconnect topologies – Cubes





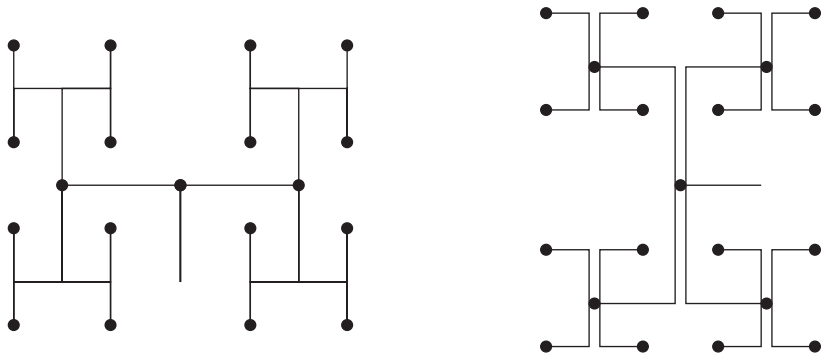
- Connect in a regular way
- Units can connect to more neighbors compared to a mesh
- Possible to establish several communication channels

Interconnect topologies – Cubes



- Practical connection of three-dimensional cube

Interconnect topologies – Trees



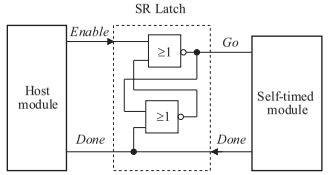
- Hierarchical division of processing elements
- Units can connect to neighbors
- Possible to establish several communication channels

Control

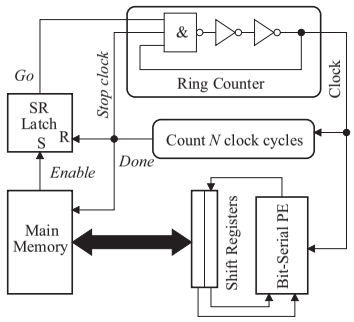
- Control timing (when things happens)
- Two main control strategies
  - Synchronous
    - Global clocks
    - Good tool support
    - Hard to synchronize large chips
  - Asynchronous
    - Handshaking
    - Need to detect how low things take
    - Limited support in tools
- Central or distributed control
- Possibly hierarchical
- GALS - Globally asynchronous locally synchronous
  - Construct the blocks using synchronous logic
  - Communicate between blocks using asynchronous logic
  - Each block can use its own clock frequency and power supply voltage

Self-times (asynchronous) architectures

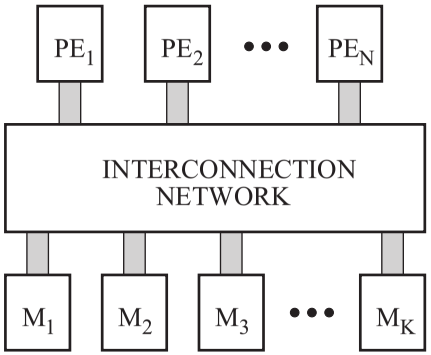
- Interface to self-timed block



- Detailed view of self-timed block



Shared-memory architectures



- N PEs and K memories
- Each PE reads/writes at most K data every N cycle
- No access conflicts as only one PE has access to the memories at any given time
- Tightly coupled as all processors have the same access time to all memories

Memory bandwidth bottleneck for shared-memory architectures

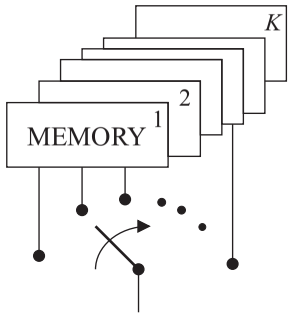
- To fully utilize N PEs (one slot for reading and one for writing)

$$T_{exe,PE} \geq 2NT_{exe,Mem} \quad (3)$$

- But $T_{exe,PE} \approx T_{exe,Mem}$
- Solutions?
  - Increase $T_{exe,PE}$
  - Decrease the number of accesses ($2N$)
  - Decrease $T_{exe,Mem}$

Reduce memory execution time

- Interleave memory L times



- Arrange memories such that L consecutive accesses fall into L different memories
- Effective memory execution time is reduced by a factor L
- $L = 2N$ for good balance

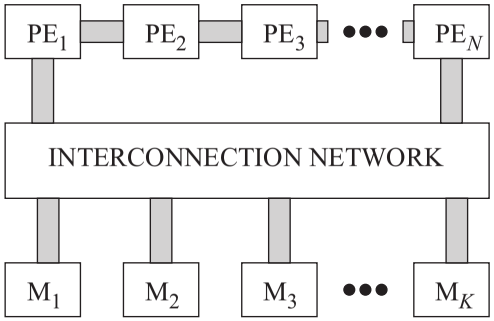
Reduce memory execution time

- Require multiple memory I/O circuits
- Memory access conflicts $\rightarrow$ all shimming delays should be at least L time units and preferably an integer multiple of L
- Quite useful and simple for $L = 2$: each cycle one memory is used for reading and one for writing, next cycle swap roles
- Similar to pipeline the memory accesses L times

Reduce communication – Broadcasting

- If the PEs operate on the same input data the number of read cycles can be decreased
  - Constraints on the scheduling, PEs sharing input data should be executed at the same time
- If outputs are written to different memories only one cycle is needed (if $N \leq K$), but N cycles if all results are written to the same memory.
  - Constraints on the scheduling

Reduce communication – Interprocessor communication



- Avoid using the memories by direct transfer of data between PEs
  - Constraints on the scheduling, operations must be placed "back-to-back"

Reduce communication – Cache memories

- Provide the PEs with a fast private memory to reduce the number of memory accesses to the main memories
- If output data is used on the same PE no need to use the main memories
- Possible to reschedule the reading/writing to free slots
- Decrease scheduling constraints when using broadcasting

Increase PE execution time – larger operations

- Make slower PEs $\rightarrow$ bad idea!
- Include more operations in PE $\rightarrow$ good idea?
- Examples:
  - Adaptor instead of adders and multipliers
  - FFT stage (BF + mult) instead of complex adders and complex multipliers
  - Complex multiplier instead of adders and multipliers
- Better balanced architecture as $T_{exe,PE}$ increases
- Often decreased number of PEs and therefore number of accesses
- Simpler scheduling due to fewer operations
- Typically lower utilization as the scheduling is less flexible

Shared memory architectures

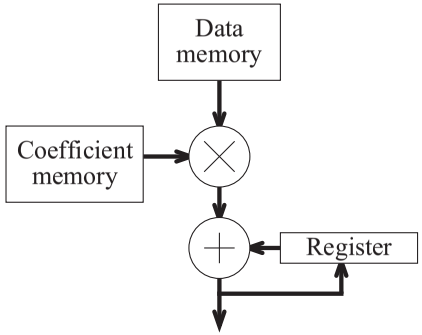
- General, most (all?) other architectures can be seen as constrained shared-memory architectures
- The initial shared memory architecture should be seen as a conceptual model
- By introducing the optimization methods above we also add constraints, mainly on scheduling of PEs and therefore memory accesses
- Many of the optimization methods above are not explicitly considered in a design
- Rather they are implicitly considered, such as not adding communication channels between memory and PE if no data will be transferred (compare with the FFT memory and PE allocation)
- Worthwhile to consider the different trade-offs if one really wants an optimal architecture

Architecture synthesis

- The shared memory architecture is general and flexible, but at the same time this makes it hard to automate
- Need to introduce constraints to simplify automation
- We will in the following consider a number of architectures which have been proposed for easy automation

Architecture synthesis – Uniprocessor architecture

- Use just one PE
- For example: (traditional) DSP processor
- Scheduling is simple $\rightarrow$ sequential ordering of operations
- Example: FIR filter



Architecture synthesis – Isomorphic mapping

- One PE per operation
- Poor utilization, especially for low sample rates
- May still need to retime and/or unfold
- Can optimize each PE for the specific task

Architecture synthesis – Vector-multiplier based implementations

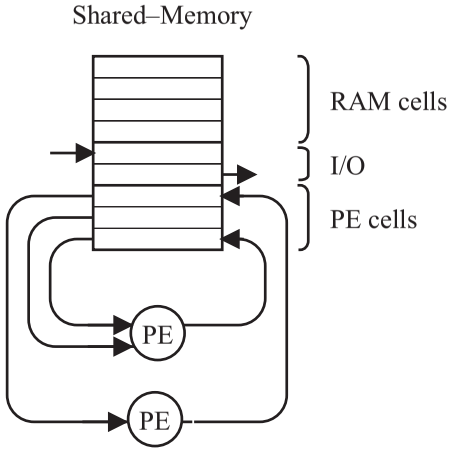
- Sum-of-products (vector multiplication) is a basic operation in many DSP algorithms

$$Y = \sum_{i=1}^{N} A_i X_i \tag{4}$$

- Maps well to e.g. (numerically equivalent) state-space representations of digital filters
- Can be efficiently implemented using e.g. distributed arithmetic (lecture 11)
- Cost approximately one to two multipliers
- Regular structure obtained where each PE computes one state which is then fed to all PEs

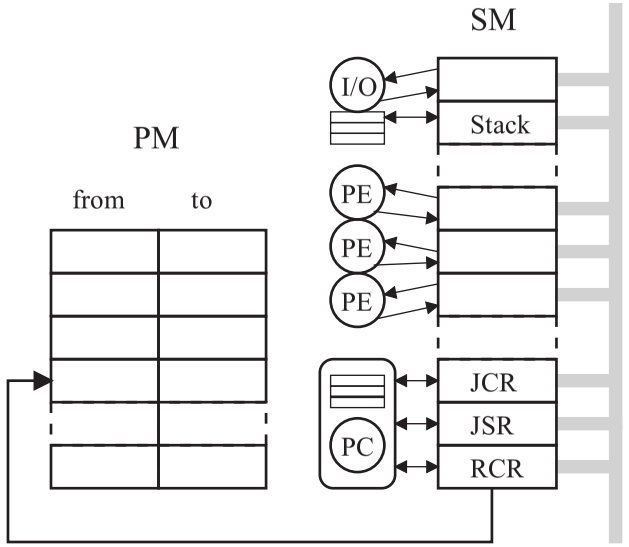
Architecture synthesis – Single instruction computer (SIC)

- Only use one instruction – memory move
- PEs and I/O mapped into a shared memory space
- Move data to a PE address to provide input data read form another to store the result etc



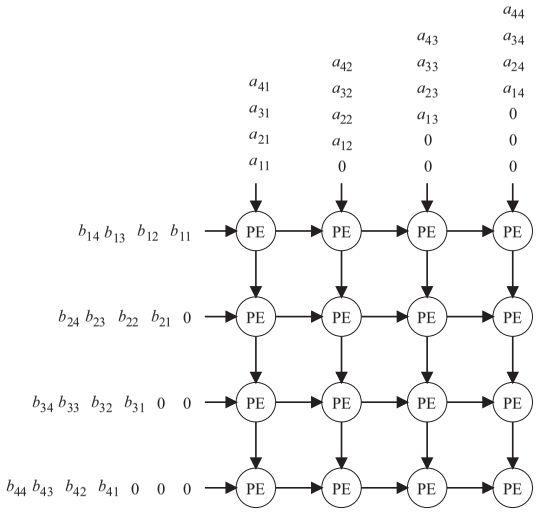
Architecture synthesis – Single instruction computer (SIC)

- More complex operations as stacks and jumps can also be handled

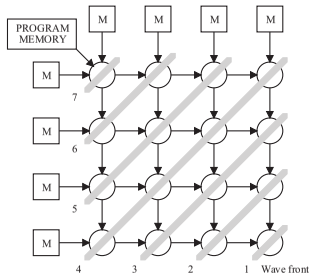


Architecture synthesis – Systolic array

- Project computations in a higher dimensional lattice space onto a lower dimensional PE space
- Projection vector determines interconnect and data movement
- Properties
  - Synchronous locally connected PEs
  - High throughput
  - High I/O-rate
- Example: matrix multiplication



Architecture synthesis – Wave-front array



- Systolic array with asynchronous communication
- Can accommodate more irregular algorithms efficiently, e.g., when $T_{exe,PE}$ is data dependent

Architectures overview

- Need to determine a suitable choice of PEs, memory, communication, and control
- Architectures can be seen as constrained shared memory architectures
- By reducing the memory bottleneck we introduce constraints on primarily scheduling
- (More constrained) Architectures more suitable for automatic synthesis