Chapter 16 - Multithreaded architectures

16.1 Introduction

Though current commodity superscalar RISC processors are very successful in exploiting functional parallelism within a chip and are able to support the built of small-scale multiprocessors, they fail in supporting large-scale parallel systems. Multithreaded architectures represent a promising new direction in processor design for supporting construction of scalable, massively parallel computers. In the current chapter a detailed study is given on the main classes and features of multithreaded architectures. Several prototype machines are also described in order to illustrate the main principles.

16.1.1 Solution for memory and synchronisation latency

In Chapter 15 the main problems of scalable parallel computers have been described. It was shown that one possible solution for tolerating memory and synchronisation latency is the introduction of threads and fast context switching mechanism among threads. In the current chapter this solution is described and explained thoroughly. This approach tolerates both the remote load and synchronisation latency problems and, therefore, plays a crucial role in the organisation of future processors. Systems which support this technique are called **multithreaded systems**. In principal multithreaded systems are very similar to multitask systems where processes are descheduled from the processor whenever they should wait for time-consuming operations like I/O or for synchronisation conditions. The general solution in these systems is that processes are temporarily suspended, i.e. their contexts are saved in a certain storage area (memory or register set) and later when the operation, they initiated, has been completed they can resume by loading the processor with the saved context. Though the context switch is usually supported by hardware the surrounding operations are managed by software (for example explicit use of semaphore operations at program level). This approach works well in coarse-grain operations where the time necessary for context switching is negligible to the time spent on the required I/O and synchronisation operations.

However, in case of remote memory access operations latency varies according to the distance between the requester and destination PEs. It can result many times in low latency which requires a fine-grain, very fast context switching. Therefore a hardware support is needed which can be applied at every remote memory access and which can guarantee that even in the case of the lowest latency the switching time is less than the latency.

Considering the synchronisation problem, a hardware mechanism is needed again that automatically recognises the situation and is able to suspend the current thread without causing idle clock cycles in the executing processor.

The processor utilisation (U) can be represented as the proportion of the useful processing time (P) and the total time (T) spent on the computation. The total time consists of the processing time, the idle time (I) when the processor is waiting for an outer condition and the switching time (S) which is used for changing the active thread on the processor.

$$U = \frac{P}{T} = \frac{P}{P + I + S}$$

The objective of any scalable parallel system is to keep U as high as possible (close to the optimum value 1). Obviously, U is high, if I and S are negligible. The purpose of the use of multithreading techniques is to reduce I without significantly increasing S. To achieve it, two conditions should be satisfied. First, a very fast context switch mechanism is needed for which S<I even for the shortest latency operations. Second, sufficiently large number of threads should be available on the processor in order to switch from a potential idle thread to an active one. Unfortunately, the maintenance of large set of threads on a processor requires a large register set and therefore a very expensive processor. To find a reasonable compromise is a crucial problem of designing multithreaded processors.

16.1.2 Design space

There are several multithreaded hardware approaches to solve the remote load and synchronisation problem. They basically differ in the computational model applied in the processor design. In chapter 1 the control flow and dataflow model have been discussed. These two models represent the two extreme points of a range of possible computational models. Between these two extremes several other models, called the hybrid models, can be found which combine features of both the dataflow and control flow models. Multithreaded machines typically belong to one of the hybrid models, though there are purely control flow based multithreaded machines, too.

Hybrid multithreaded machines can be classified according to their position in the scale of computational models. Macro dataflow architectures are the closest ones to the original dataflow machines. Decoupled architectures are based on the control token concept and represent the next step towards control flow machines. Finally, RISC-like hybrid architectures are the closest relatives of the original control flow machines. They

work on a parallel control flow model based on parallel control operators. The possible computational models are compared in the next section.

The other important design decision is related to the granularity of thread interleaving. The two main directions are the application of fine-grain and coarse-grain approaches. The fine-grain concept means that threads are interleaved at the level of clock cycles inside the processor pipeline in order to ensure maximal processor utilisation. Fine-grain multithreaded architectures are optimum as far as there are sufficiently large number of active threads available inside the processor to keep the processor pipeline busy. However, they suffer poor single-thread performance and low processor use when there is not enough parallelism in the program. To avoid this problem coarse-grain multithreaded machines execute only one thread at a time. When the active thread encounters remote load or failed synchronisation, thread switching occurs and the processor executes the new thread while the preceding one is suspended. Notice that coarse-grain multithreading is a more conventional approach than fine-grain multithreading. The former one can be derived from commodity superscalar RISC processors as an evolutionary modification of these chips while the latter one is typically progressed from the revolutionary dataflow approach.

Multithreaded parallel computers can be further classified according to their memory system. Some of them contain physically shared memory, like classic shared memory architectures, while others have only physically distributed memory and work as distributed shared memory systems. In both cases an important aspect of designing large-scale multithreaded machines is the employment of coherent cache memories. Cache memories are able to reduce the memory access latency at cache hit, but in order to maintain cache coherency they increase latency at cache miss. (The effect of cache memory will be analysed in detail in Chapter 18.) These contradictory features of cache coherent systems require a thorough investigation of how to combine multithreading with coherent caches. It is worth mentioning that in order to reduce the negative effect of the cache coherency problem recent scalable parallel computers apply some kind of multithreading support even if they are not particularly designed as multithreaded computers.

The fourth determining design decision should take into consideration the optimum number of threads to be handled by hardware inside a processor. Usually, fine-grain thread interleaving without cache support requires a large number of threads to keep the processor busy all the time. In coarse-grain multithreaded architectures supported by coherent cache it is sufficient to handle only a very limited number of threads inside the processor although, the cache coherency problem complicates the processor and system design. Between these two extremes there are various possibilities to tune the number of threads.

The design space of multithreaded machines is shown in Figure 1. The multithreaded machines studied in this chapter are classified on the basis of the design space described above and their classification tree is depicted in Figure 2.

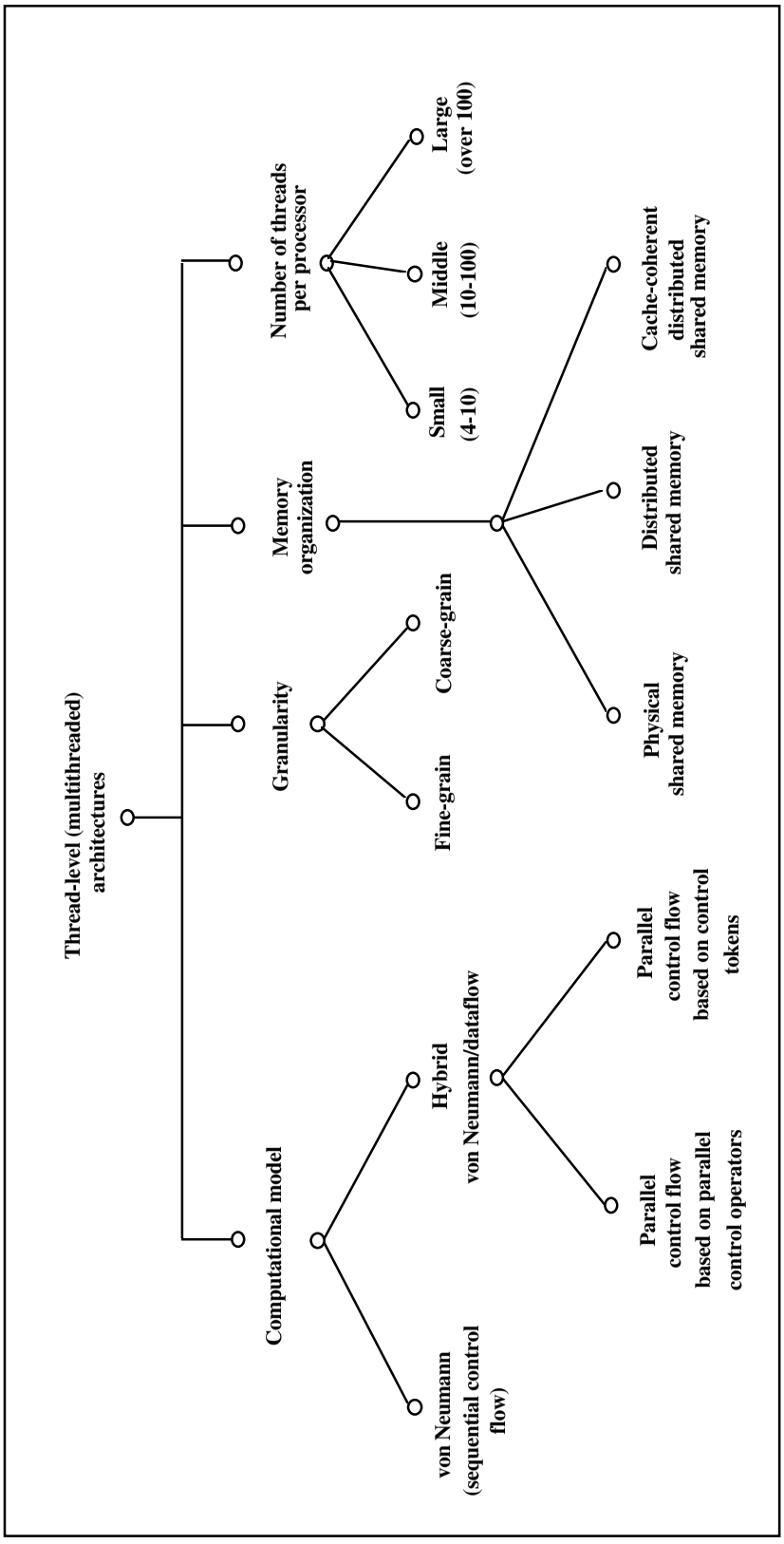

Figure 1. The design space of Thread-level (multithreaded) architectures

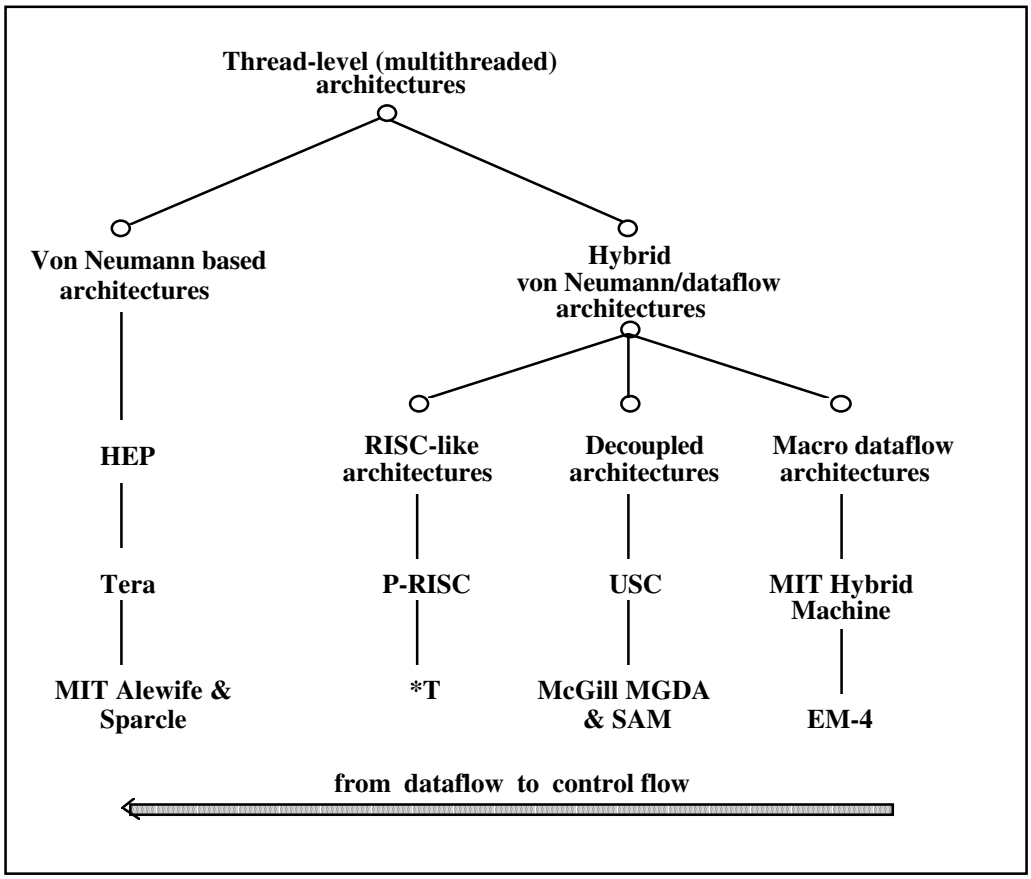

Figure 2. Classification of multithreaded architectures

16.2 Computational models

From the point of view of multithreaded architectures three computational models are relevant:

- von Neumann model
- dataflow model
- hybrid von Neumann/dataflow model

Though their main characteristics are discussed in Chapter 1, here we concentrate on their operational semantics in order to explain how they are related to the various classes of multithreaded machines.

16.2.1 Sequential control flow (von Neumann)

In the von Neumann computational model the flows of control and data are separated. Instructions of the program should be executed sequentially determined by the textual order of instructions in the program text. Any deviation from this textual order should be explicitly defined by special control instructions like GOTO, JUMP, CALL, etc. which are generated either by the programmer or by an intelligent compiler based on

program analysis. The hardware support for this sequential execution scheme manifests in the program counter which is automatically incremented after executing ordinary instructions or explicitly set by the special control instructions.

Data are stored in reusable memory cells (or registers). The flow of data is determined by references to these memory cells and it has no effect on the execution order of instructions. Accordingly, the von Neumann model is often called sequential control flow model too, since, it is the control flow which determines the instruction execution sequence.

16.2.2 Dataflow

In the dataflow model the control is tied to the flow of data. The textual order of instructions in the program text has no effect on the execution order. An instruction is executed if all the data needed for the execution are available. There is no concept of shared, rewritable data storage. Data are in continuous flow independently of reusable memory cells and their availability initiates instruction executions. Since data can be available for several instructions at the same time, these instruction can be executed in parallel.

The potential parallel computation is reflected by the data dependency or **dataflow graph** whose nodes are the instructions of the program and the arcs represent data dependency between instructions (see Figure 3.). Data move on the arcs of the graph in form of **data tokens** which contain data values and status information. The asynchronous, parallel computation is determined by the **firing rule** which is expressed by means of tokens: a node of the dataflow graph can fire if there is a token on each input arc. If a node fires it consumes the input tokens, performs the associated operation and places result tokens on the output arcs.

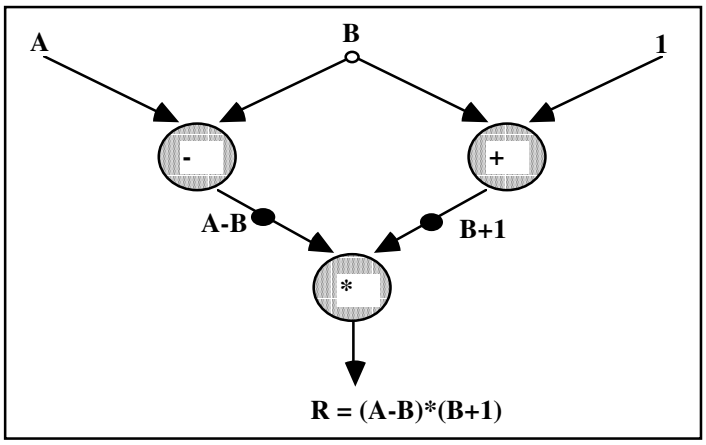

Figure 3. Dataflow graph to compute R=(A-B)*(B+1)

The advantage of the dataflow concept is that nodes of the dataflow graph are self-scheduling based on the firing rule. However, the hardware support to recognise the

availability of necessary data is much more complicated than the simple program counter of von Neumann machines. The different generations of dataflow architectures reflect the various solutions applied for solving this problem.

16.2.3 Parallel control flow (hybrid) models

Parallel control flow models represent transition from the sequential control flow model to the parallel dataflow one. The parallel control flow model that is based on **control operators** is closer to the sequential control flow model, while the **control tokens** based model resembles to the dataflow one in several ways.

Parallel control operators

Like the sequential control instructions of the von Neumann model, parallel control operators are used to control the execution order in programs. However, meanwhile the sequential control instructions simply alter the textual order, the parallel control operators are able to change the number of threads in the computation. The FORK instruction serves for creating new threads and defining the starting instructions of new threads. The JOIN instruction reduces the number of threads by defining points in the program execution where two threads should rendezvous and only one of them can continue, the other is killed.

Computation is based on the von Neumann model inside a thread. Communication among threads is realised by reusable memory cells (or registers) which require extra synchronisation care (for example semaphores). Notice that the flows of control and data are still strongly separated. The previously shown simple example and its execution scheme based on the parallel control flow model is depicted in Figure 4.

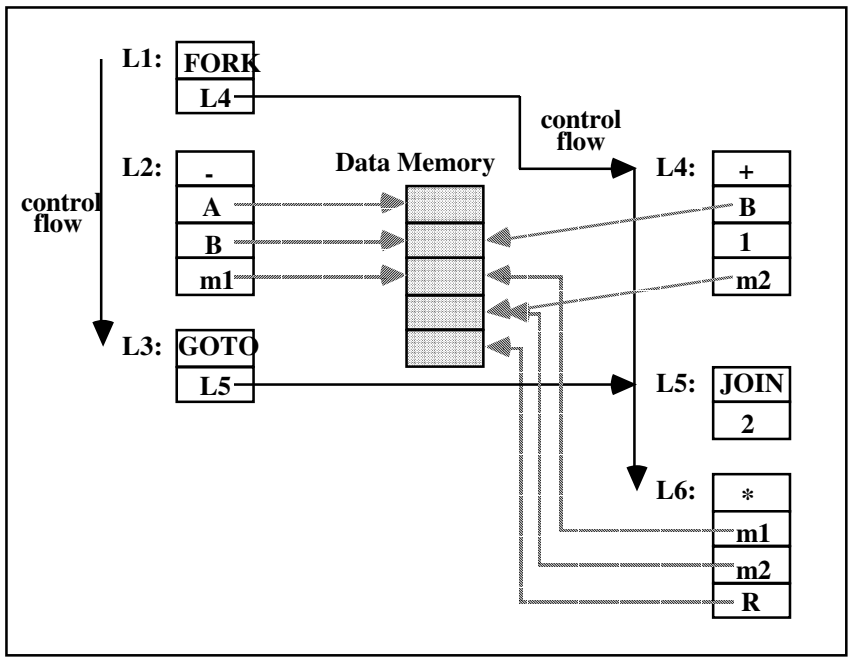

Figure 4. Parallel control flow model based on control operators to compute R=(A-B)*(B+1)

Control tokens

The parallel computational model based on control tokens resembles to the dataflow model in many ways. The computation is specified by the data dependency graph like in the dataflow model. However, the physical representation of this graph is different from the representation applied in the dataflow model. Here the concept of rewritable shared data storage of control flow machines still exists and hence so-called **control tokens** move on the arcs of the graph instead of data tokens. According to the graph each instruction specifies its successor instruction and after execution it sends control tokens to its successors. An instruction can be executed if it has received all the required control tokens (each input arc contains a control token).

Obviously there is a strong analogy between the dataflow model and the parallel control tokens based model, although, in the latter model data and control are separated like in the von Neumann model. This separation means that reusable memory cells are still maintained for data handling. However, if their access is synchronised by the data dependency graph, other synchronisation methods, like semaphores, are not needed for their access. The simple example and its execution scheme based on the control tokens is shown in Figure 5.

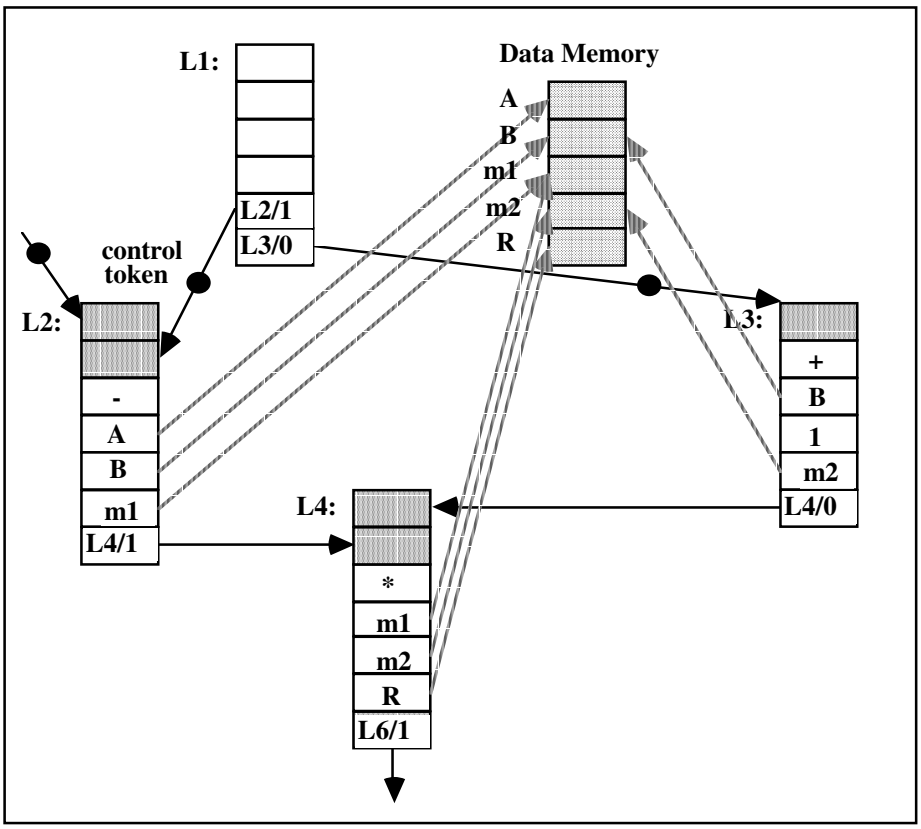

Figure 5. Parallel control flow model based on control tokens to compute R=(A-B)*(B+1)

16.3 Neumann based multithreaded architectures

16.3.1 HEP

Roots of the multithreaded architectures go back to the Denelcor HEP machine which was a revolutionary novel design in the second half of the seventies. The HEP was the very first parallel architecture designed with objectives of tolerating memory latency and synchronisation problems. It represents a fine-grain interleaving of threads where the processor pipeline contains eight stages and at each time step a new thread enters the execution pipe. Threads are taken from the Process Status Word Queue (PSW Queue) which is the fundamental element of the Control Loop (Figure 6.). After a thread is taken from the head of the PSW Queue, the current instruction of the thread is fetched from the Program Memory and the operands from the registers. Finally the instruction is executed by one of the functional units and the thread with an incremented instruction pointer is placed back to the end of the PSW Queue. The minimal length of the queue is eight in order to be able to insert a different thread to the pipeline at every clock step. As a result, threads are interleaved at the instruction level.

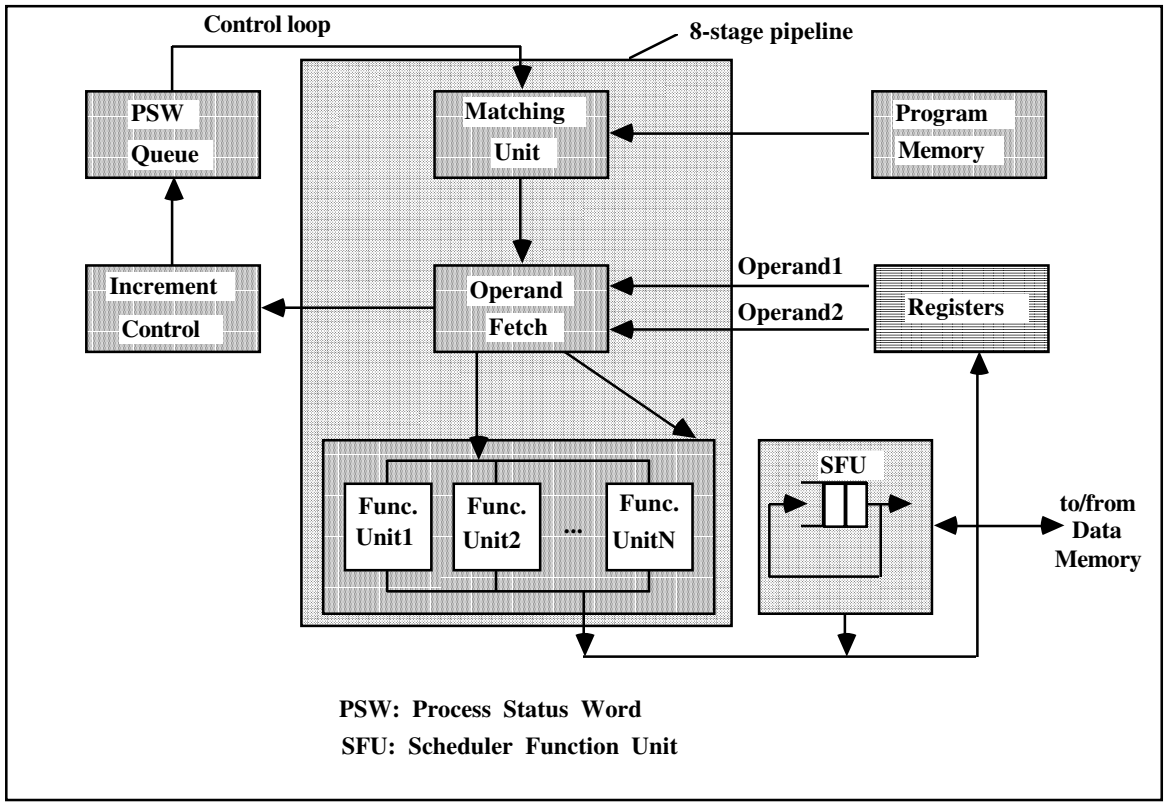

Figure 6. Structure of the HEP processing element

Memory latency toleration is solved by the Scheduler Function Unit (SFU) and by extending the Data Memory words with a Full/Empty bit. Attempting to read a word whose Full/Empty bit is in the state Empty, results in the suspension of the current thread. In order to avoid the useless circulation of such a thread in the Control Loop, the PSW of the thread is moved from the Control Loop to the SFU. The PSW of the suspended thread circulates here and reissues the memory request until the required memory access can be fulfilled. At this time the thread is placed back to the Control Loop and can continue its program execution. Notice that suspended threads do not waste execution stages in the processor pipeline if they can be replaced by other active threads of the Control Loop. Maximum 64 PSWs (threads) can be handled at each processor in the HEP machine.

Synchronisation among threads is realised via register communication. In each processor there are 2048 registers supplied with Full/Empty/Reserved bits for synchronisation purposes. An unsuccessful attempt of communication via a register results in a NO-OP operation in the processor pipeline and the thread is placed back to the PSW Queue without updating its program counter. In this way thread synchronisation is realised by a busy-wait way but the processor is not blocked. Other threads can actively use the processor, meanwhile, the waiting thread causes bubbles in the processor

pipeline as long as it is taken from the PSW Queue and the synchronisation condition is not satisfied yet.

The HEP machine was a shared memory multiprocessor. It had up to 16 processors and up to 128 data memory modules which were connected in a "dance-hall" configuration via a high-speed switching network.

Though the HEP represents a revolutionary step in the processor design it has several limitations:

1. Threads can have only one outstanding memory request.

2. Though busy-waiting of threads on synchronisation conditions does not block entirely the processor, the insertion of bubbles into the processor pipeline leads to loss of processor time.

3. Maximum 64 threads per processor can be used which represents a serious impediment to the software development on the HEP.

4. When the available parallelism is limited in a program, i.e. the number of available threads is less than eight, the full utilisation of the processor can not be assured.

16.3.2 Tera

The processor architecture of the Tera machine is a modern version of the architecture of the Denelcor HEP processor. It is based on the same fine-grain multithreading principle, only the number of applicable threads per processor has been doubled due to the improved available technology. In order to make context switching extremely fast, each of the 128 threads has 41 registers (64-bit each) in the Tera processor.

The greatest difference between the Tera and the HEP is in the interconnection of processors and memory modules. While the HEP was a physically shared memory multiprocessor, the Tera machine is a distributed shared memory system. It can have up to 256 processors, 512 memory modules, 256 I/O cache units, 256 I/O processors and 2816 communication nodes which have no functional units or resources attached. The interconnection network contains 4096 nodes arranged in a sparse 16x16x16 toroidal mesh.

The Tera machine is a state-of-the-art distributed shared memory system in which processor design is based on the multithreading principle in order to tolerate memory and synchronisation latency. The clock period is less than 3 nanoseconds and the architecture is scalable essentially without limit. The multithreaded Tera processor has the same advantages and drawbacks as the HEP processor.

16.3.3 MIT Alewife and Sparcle

The MIT Alewife machine is a large-scale cache coherent distributed shared memory (CC-NUMA) system. The maximum 512 processing nodes are arranged in a two-dimensional mesh topology. Each node consists of a Sparcle processor, 4-Mbyte main memory, 64-Kbyte cache, 4-Mbyte local memory, floating point unit, communication and memory management unit (CMMU), and a network router/switch. The 4-Mbyte main memory is part of the distributed shared memory, while the 4-Mbyte local memory contains the cache coherence directory, code and local data. The structure of the Alewife machine is shown in Figure 7.

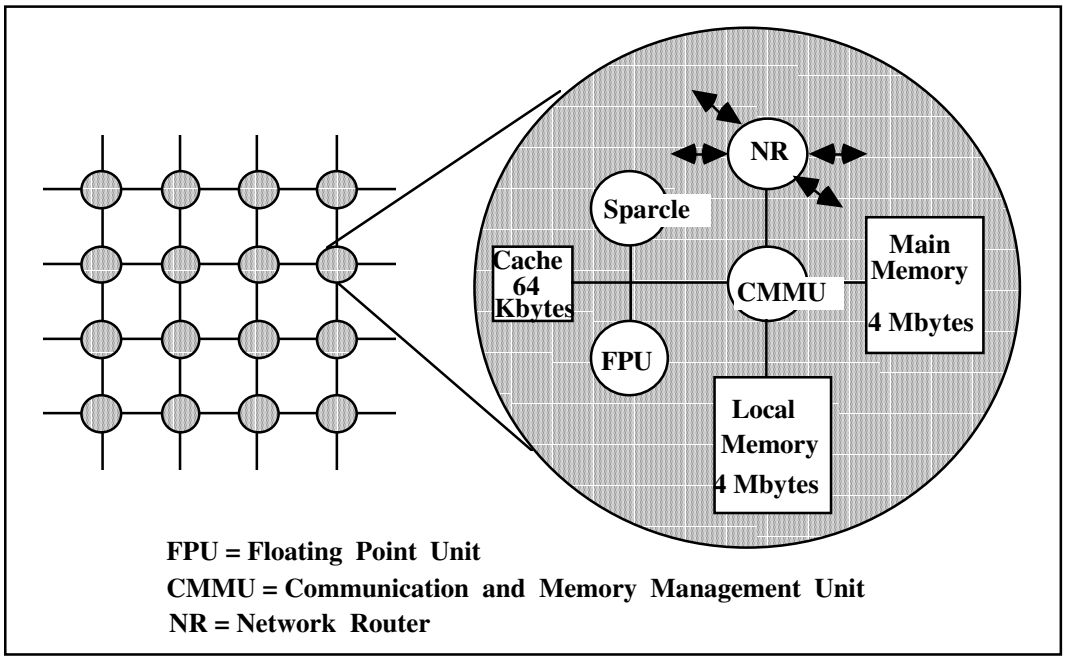

Figure 7. Structure of the Alewife machine

The MIT Alewife is one of the recent state-of-the-art scalable parallel computers that combine the main features of this kind of machines:

- physically distributed memory
- logically shared memory
- hardware-supported cache coherence
- hardware-supported user-level message-passing
- multithreading

The main focus point in the design of Alewife was the integration of multithreading capabilities with hardware-supported cache coherence. Accordingly, here we concentrate on the multithreading property of the Alewife machine by explaining in detail Sparcle, the processor chip of Alewife.

The Sparcle processor

Sparcle is an evolutionary extension of the Sparc architecture in order to solve the problems of scalable parallel computers:

1. memory latency tolerance by fast context switching
2. fine-grain synchronisation
3. efficient user-level message-passing

Fast context switching

In Sparc there are 8 overlapping register windows which are used in pairs by Sparcle in order to represent 4 independent, non-overlapping contexts. Three of them are used for user threads and the fourth for traps and message handlers. Each context contains 32 general purpose registers and the following three special purpose registers:

- PSR (Processor State Register)
- PC (Program Counter)
- nPC (next Program Counter)

User threads can be in one of the four states:

- active
- loaded
- ready
- suspended

The CP (Context Pointer) register points to the active context. A thread is active if its context is pointed by the CP. A thread is loaded if its context is stored in one of the three thread contexts. A loaded thread is able to continue its progress if it becomes active. A thread is suspended if it should wait on a synchronisation condition. Ready threads are not suspended and not loaded yet. The possible states and positions of threads are shown in Figure 8. Thread switching is fast if it takes place between the active and one of the loaded threads. In this case it contains only two main steps:

1. changing the value of the CP register
2. flushing the processor pipeline.

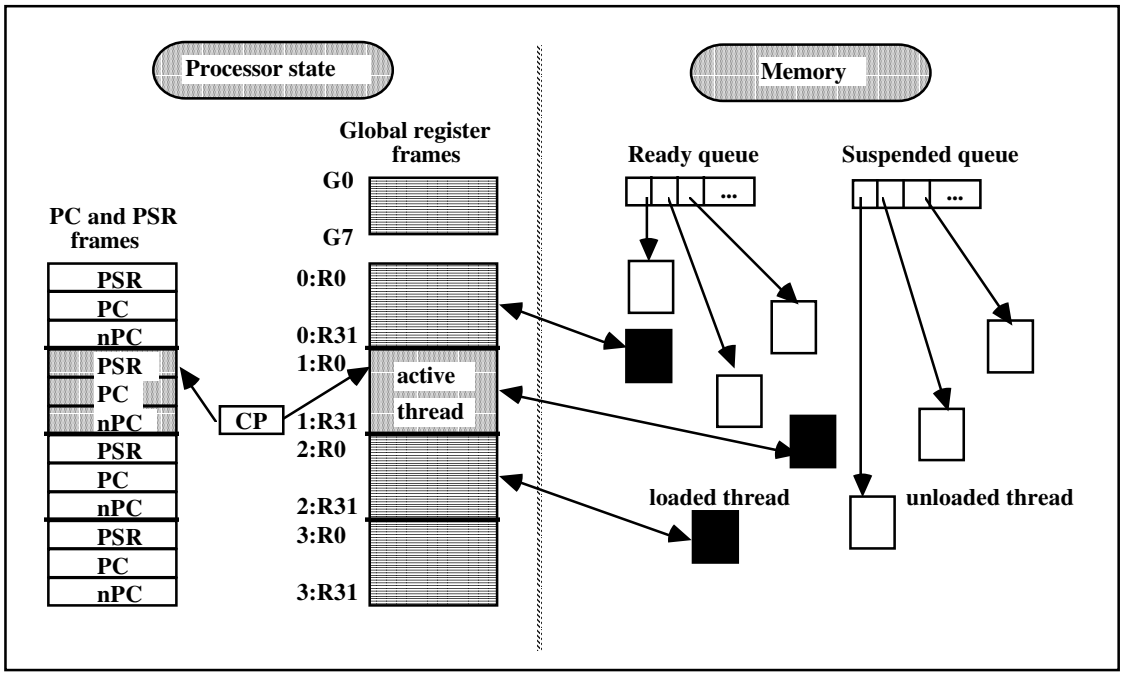

Figure 8. Thread states in Sparcle (Agarwal & al. 1993)

Even the fastest thread switch requires 14 cycles which means a relatively large context switch overhead comparing to the fine-grain multithreading architectures. That is the reason why cache support is used in the Alewife machine: to reduce the frequency of thread switches.

Support for fine-grain synchronisation

Both data-level and control-level synchronisations are available in the Alewife machine. The data-level synchronisation is very similar to the I-structure introduced in the MIT Tagged-Token Dataflow Machine (see Section 16.4.2). Two versions, called the J-structure and L-structure, are applied in the Alewife machine. They are data structures supporting producer-consumer style synchronisation by extending memory words by a special state bit (Full/Empty bit). The difference between them comes from the types of operations they support. On J-structure only two operation types are allowed: write and nonlocking read, while on L-structures a third one, the locking read is performable too.

When a J-structure or L-structure is allocated the state bits are in Empty state for all elements of the data structure. The content of each element is -1 showing that no reader thread waits for a write operation. A read operation in this state results in the suspension of the requester thread by putting it into a suspension queue and a pointer to this queue is placed in the corresponding element. A write operation on such an element sets the state bit to Full and releases all the waiting threads which after becoming active can read the element. A locking read empties the element after reading its value, while nonlocking read leaves the element in Full state.

Although both the HEP and Alewife machines use the Full/Empty memory tag for synchronising thread communication, there exists a significant difference. In the HEP an unsuccessful attempt to access a memory cell results in a repeated issue of the store/load operation which can lead to an unnecessarily increased traffic on the communication network. In the Alewife machine threads are suspended and placed in a suspension queue which is originated from the memory cell to which the memory access was failed. When the synchronisation condition (empty for store and full for load operation) is satisfied the suspended threads are invoked and placed back to the Ready queue. Notice that the J- and L-structures behave like semaphores in multitask systems. The difference is that manipulation on thread suspension and ready queues is done by hardware while semaphore queues are operated by software.

Control-level synchronisation is based on the **future** concept.

future X

declares that the evaluation of X and the execution of the thread containing the future instruction can continue in parallel. In order to pass the result of the evaluation of X a **placeholder** is created. Whenever a thread attempts to read the value of X before its evaluation is completed, the task is suspended on the placeholder of X. When the value of X is produced all the threads waiting on the placeholder will be released. The implementation mechanism is similar to the data-parallel synchronisation, the low bit of placeholders is used as state bit.

Communication and memory management unit

Besides the Sparcle processor, the communication and memory management unit (CMMU) should play an active role in supporting latency tolerance and synchronisation. For the sake of latency tolerance the CMMU should be able to handle split-phase memory transactions where the memory request is separated from the response. In the case of cache miss the cache controller initiates the context-switch trap and at the same time sends a request message to the remote processing element holding the requested data item.

For the full/empty bit synchronisation an extra storage area should be reserved in the cache system. A notable feature of the Alewife is that the state bits are stored together with the cache tags which has several advantages. The physical memory applied for cache data can be an ordinary one. The access to the cache tags is faster than to the cache data since on one hand the tag file is smaller and on the other hand crossing chip boundaries is not needed.

Assessment of the Alewife architecture

The most important design aspect of the Alewife machine is the combination of coherent caches and multithreading. The advantage of this combination is that the relatively large context switch overhead does not cause significant performance reduction. Processor utilisation depends on the product of context switching overhead and switching frequency. Due to the presence of cache the need for remote access is relatively rare and hence the switching frequency is low.

The Alewife and Sparcle projects have demonstrated that contemporary RISC processors could easily be modified towards supporting scalable parallel computers particularly if large coherent cache (greater than 64 Kbytes) is allocated for each processing node.

16.4 Dataflow architectures

The execution mechanism of dataflow architectures is defined by the dataflow computational model described in Chapter 1 and Section 16.2.2. Dataflow architectures can exploit both instruction and procedure level parallelism depending on the complexity of functions associated with nodes of the dataflow graph (DFG). Instruction level parallelism appears when nodes of the DFG represent single instructions. Procedure level parallelism is exploited when nodes of the DFG are associated with complete functions. Three generations of dataflow architectures can be distinguished according to their approaches for solving the problem of token matching.

1. static dataflow architectures
2. tagged-token dataflow machines
3. explicit token-store architectures

The following sections explain the properties of these generations in detail.

16.4.1 Static dataflow architectures

The first generation of dataflow architectures are based on the 1-token/arc principle which is governed by the **static firing rule**: A node can fire if there is a token on each input arc and *there is no token on its output arcs*. A token is represented by a 3-tuple:

<v, <f,n>, a>

where v is the data value conveyed by the token, f represents the current function activation, n is the target node inside the function and a represents the destination arc of the target node. Tokens of a target node match if their <f,n> fields (called the tag) are the same.

The structure of a typical static dataflow architecture is shown in Figure 9. The Activity Store consists of two memory units: the Instruction/Data Memory and the Control Memory. A node of the dataflow graph is represented by a frame in the Instruction/Data Memory. Each frame has the following structure:

- operation_code
- operand_1
  :
- operand_N
- target_node/destination_arc_1
  :
- target_node/destination_arc_K

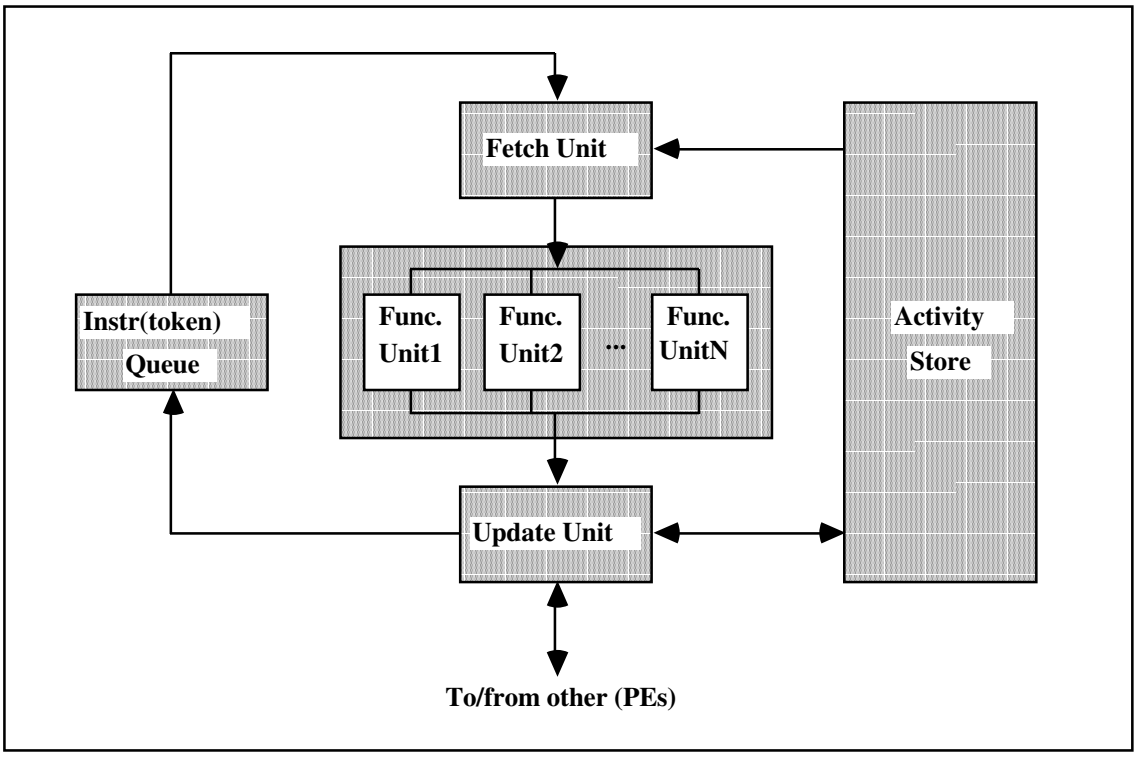

Figure 9. Structure of a typical static dataflow PE

A Control Memory frame belongs to each Instruction/Data Memory frame. The Control Memory frame contains presence bits for all operands and target_node/destination_arc fields. The presence bit is set for an operand if the operand is available (an input token arrived to the input arc of the associated node of the dataflow graph) and for a target_node/destination_arc field if the associated arc does not contain a token. A node can fire (the corresponding instruction can be executed) if all the presence bits in the Control Memory frame are set. When this situation is recognised by the Update Unit, it places an instruction packet to the Instruction Queue. Based on the instruction packet and the contents of the Activity Store, the Fetch Unit composes an operation packet and sends it to one of the idle Functional Units. After executing the

required operation the Functional Unit creates a result packet and sends it to the Update Unit which updates the Activity Store according to the results.

The parallelism achievable by this generation is strongly limited. For example, when a complex function is to be executed in a loop, the function invocations belonging to different loop cycles should be executed sequentially even if they are independent.

16.4.2 Tagged-token dataflow machines

Tagged-token dataflow machines were introduced for eliminating the problem of static dataflow machines. They do not restrict the number of tokens on arcs of the dataflow graph and as a result they do not limit the achievable parallelism. However, this concept raises a new problem: tokens belonging to different activations should be distinguished when tokens are matched. For example, a node with two input arcs must consume two tokens belonging to the same cycle of a loop iteration. This is achieved by introducing a **token colouring** scheme. The colour is an index which is carried by the token and, for example, in case of loops it identifies the cycle of the loop the token belongs to.

In the tagged-token model the structure of a token is more complex than in the static model:

<v, <f,n,c,i>, a>

where c specifies a code block or loop body within function activation f and i (the index) represents the colour. The **firing rule** of the tagged-token model becomes: A node can fire if there exists a colour "i" for which tokens of colour "i" are available on each input arc. Due to the colour, the matching of tokens becomes a more complex operation than it was in the static dataflow machines.

The typical structure of tagged-token dataflow machines is shown in Figure 10. Notice that a new unit, the Matching Unit has to be introduced into the processor pipeline in order to perform coloured token matching. The Matching Unit is a special memory containing tokens which wait for their partner token of the same colour. The Matching Unit receives tokens from the Token Queue and checks if their partner token is already available in the matching store. If the partner token is missing, the received token is placed in the matching store. If the partner token is present, the Matching Unit removes the matching token from the store and passes the two matching tokens to the Fetch Unit. Based on the common tag, the Fetch Unit finds the enabled instruction in the Instruction/Data Memory and composes an operation packet which is sent to one of the idle Functional Units. Finally, the Form Token Unit creates result tokens and places them into the Token Queue.

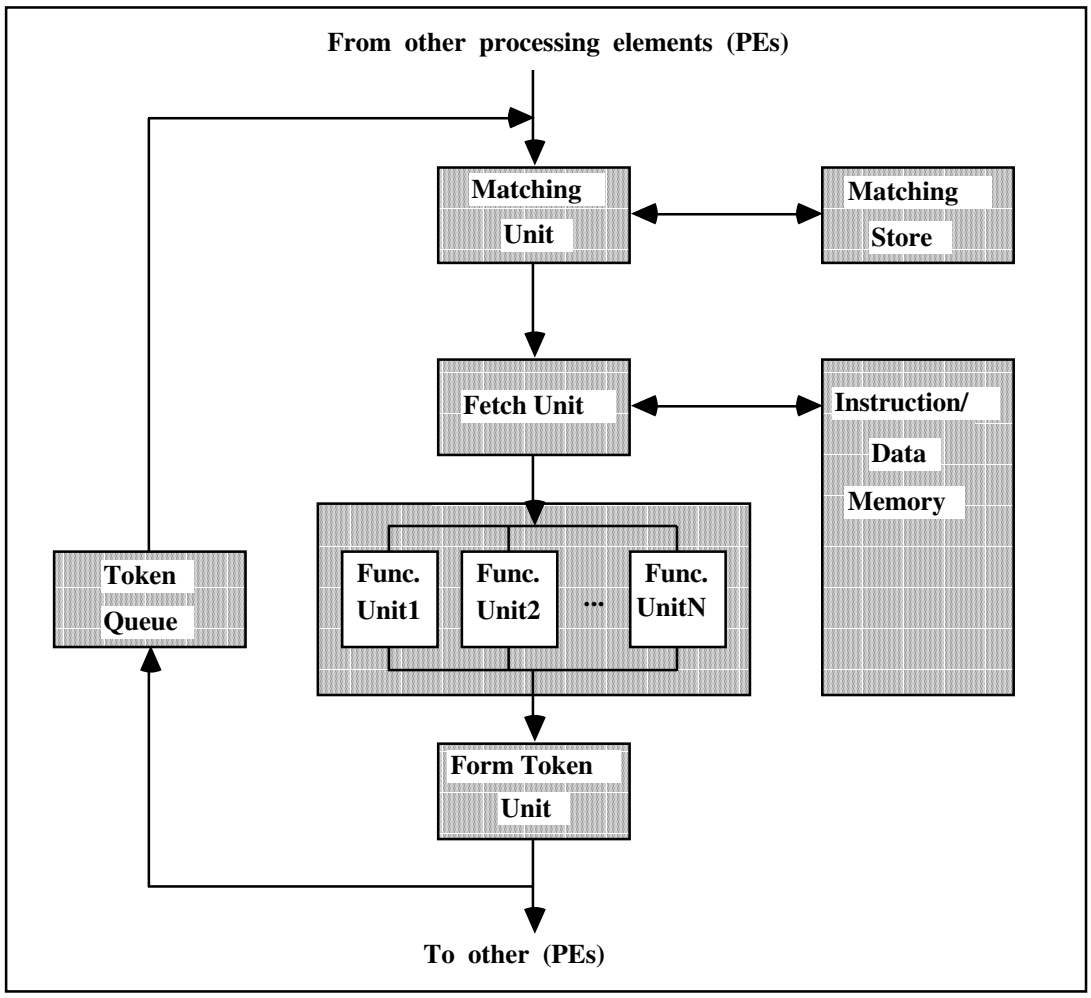

Figure 10. Structure of a typical tagged-token dataflow PE

Another problem to be solved by dynamic dataflow machines is array handling. Since there is no concept of reusable memory cells (or variables) in dataflow models, the single assignment principle is valid even for data structures like arrays. This implies that every time an element is updated the whole array should be copied, which is prohibitively expensive. To tackle the problem several schemes have been proposed:

1. Heaps
2. I-structures
3. Token relabeling
4. HDFM-arrays

The most successful one, particularly from the point of view of multithreading, is the **I-structure** concept introduced by Arvind (Arvind and Thomas 1980). The I-structure storage consists of two parts: Data storage and Deferred read lists (see Figure 11). In the Data storage each memory cell is extended by Presence bits which indicate the status of the cell:

Present: A store operation has already placed the datum in the cell.
Absent: Neither store nor load operation has been executed.

Waiting: One or more load operations have been executed, but no store operation.

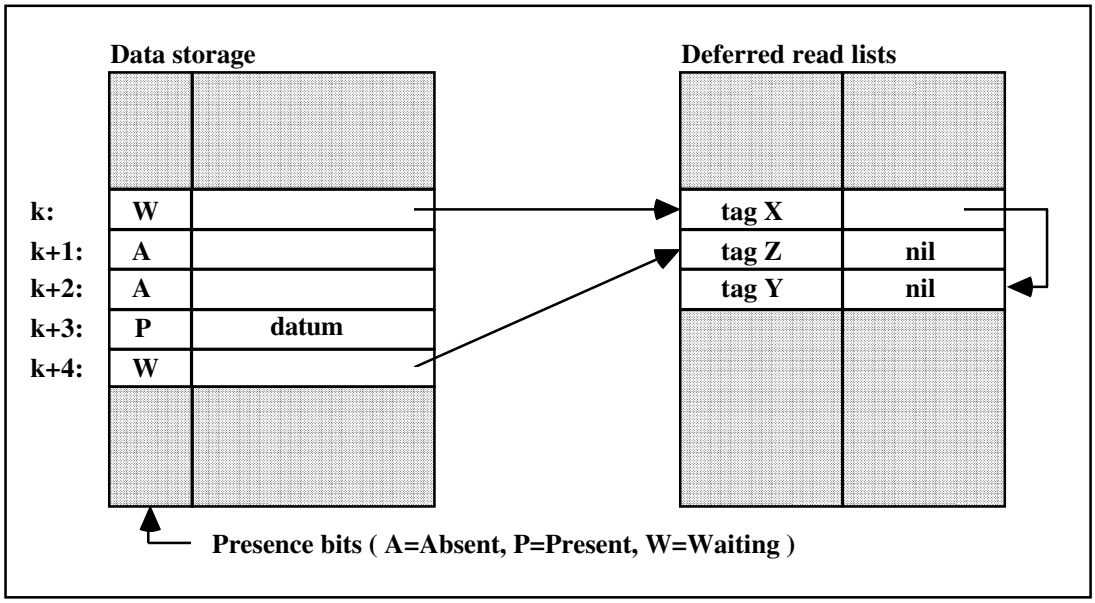

Figure 11. Organization of the I-structure storage

When a cell is in Waiting state the read requests are deferred and placed in the Deferred read list whose pointer is stored in the memory cell. When the store operation is performed on such a Waiting cell, all the pending read requests are removed from the Deferred read list and the newly-arrived datum is sent to their target instructions. The memory cell is updated by storing the datum and modifying the state to Present. Notice that the load operation implemented by the I-structure is a **split-phase transaction.** The dataflow processor can issue multiple, overlapped memory requests and can tolerate out-of-order responses. Accordingly, tagged-token dataflow architectures, supported by I-structure memory, can tolerate latency of remote memory access and therefore represent a good candidate for scalable parallel architectures. It is worth mentioning that the I-structure memory resembles to the tagged memory concept of HEP. However, in the HEP unsatisfiable memory requests lead to busy-waiting condition since the concept of Deferred read list is missing there.

Comparing the static and tagged-token dataflow machines there are advantages and drawbacks on both sides. The tagged-token model is favourable to the static model since it enables to exploit much more parallelism and, supported by I-structure, can tolerate memory latency. However, in the machine organisation the price for this enhanced parallelism appears in the extra Matching Unit which requires either a large-capacity associative memory which is too expensive or a pseudo-associative matching mechanism that needs several memory accesses which renders unbalanced the processor pipeline of the tagged-token dataflow architectures.

16.4.3 Explicit token-store architectures

A significant progress in dataflow architectures was the invention of the explicit token-store mechanism that eliminated the main drawback of tagged-token dataflow architectures. The main idea of this architecture type is the perception that tokens of the same colour belong to the same activation block and, therefore, a separate memory segment can be created for them when the block is activated. The **Frame Memory** was introduced for this purpose. Whenever a block is activated a frame is allocated in the Frame Memory to hold the tokens belonging to the current block activation. The colour of these tokens is represented by the frame pointer (FP) of the activated block. Similarly, the f, n, c fields of the <f,n,c,i> tag are replaced by a single pointer, the instruction pointer (IP) which always points into the code block of the activated function in the Instruction Memory. The representation of a token in the explicit token-store architectures becomes:

<v, <FP,IP>>

The instruction format in the Instruction Memory is as follows:

- operation code (OP)
- frame memory index (I)
- destinations (D)

Here the frame memory index is an offset value to the rendezvous slot in the Frame Memory and the destination (D) is a relative index into the instruction memory pointing to a destination instruction. A simple example for coding a dataflow graph and tokens on its arcs is shown in Figure 12.

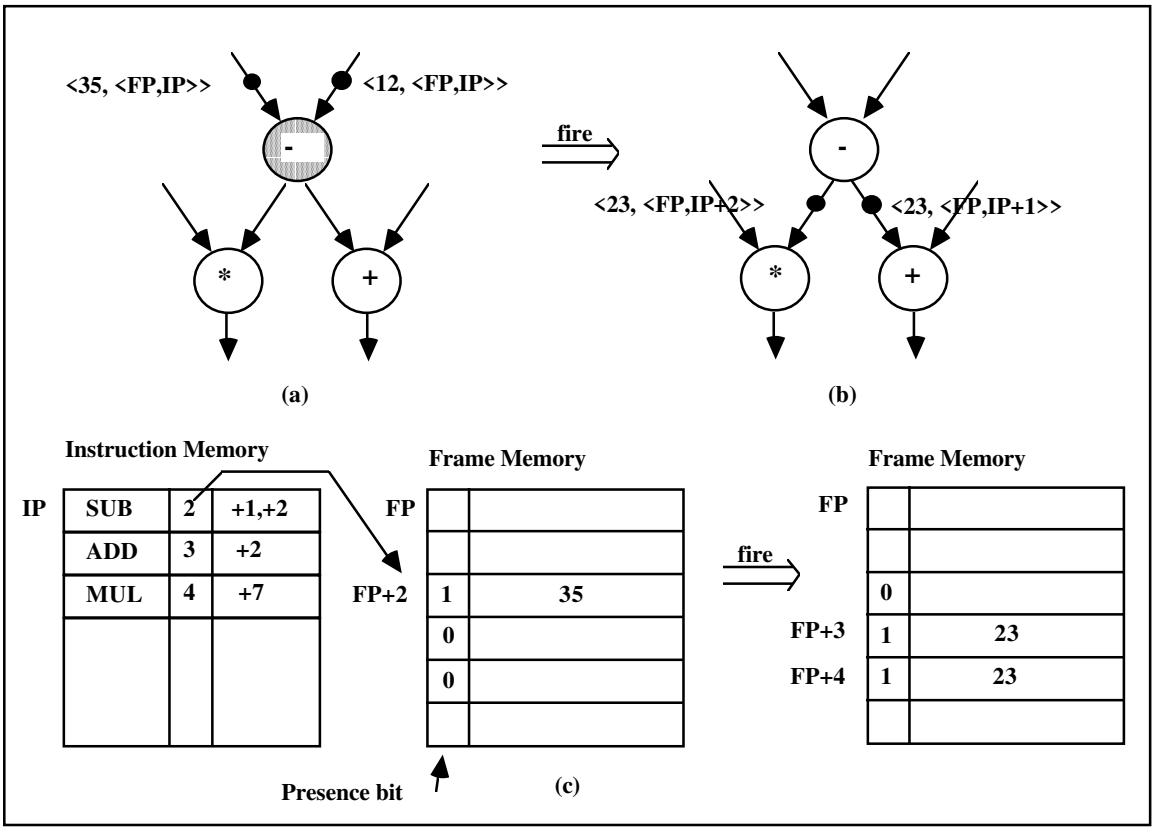

Figure 12. Coding in explicit token-store architectures

Each word of the Frame Memory is extended with a Presence bit similarly to the data memory of the HEP machine. Like in the tagged-token dataflow architectures nodes can have maximum two input arcs. When a token <v1, <FP,IP>> arrives to a node the Presence bit of the Frame Memory location FP+(IP.I) is checked. If the Presence bit is reset, the partner token has not arrived yet and hence the value field of the current token should be temporarily stored in this Frame Memory location and the Presence bit should be set indicating that the first token is already available.

(FP+(IP.I)).value := v1
(FP+(IP.I)).presence := 1

This case is shown in Figure 12. after arriving the first token <35, <FP,IP>> to the left input arc of the SUB node.

If a token <v2, <FP,IP>> arrives to a node for which the value v1 of the first token is already stored, the instruction representing the node (at location IP of the Instruction Memory) can be executed with the two values v1 and v2. At this time v1 is fetched from the Frame Memory and the Presence bit is reset and an instruction packet <v1, v2, FP, IP, IP.OP, IP.D> is passed to the Functional Unit which executes the requested operation and passes the result to the Form Token Unit. Figure 12. illustrates what happens when the second input token for the SUB node arrives. The SUB operation becomes firable and

after its execution the result is passed to the ADD and MUL nodes which require the input token in the position of FP+3 and FP+4, respectively.

The typical structure of such an explicit token-store architecture is shown in Figure 13. Notice that the token matching function has become a short operation which makes it possible to introduce a several-stage well balanced processor pipeline. Indeed, in the Monsoon explicit token-store architecture an 8-stage pipeline has been applied which significantly increases the processor performance compared to tagged-token dataflow architectures (Papadopoulos and Culler, 1990).

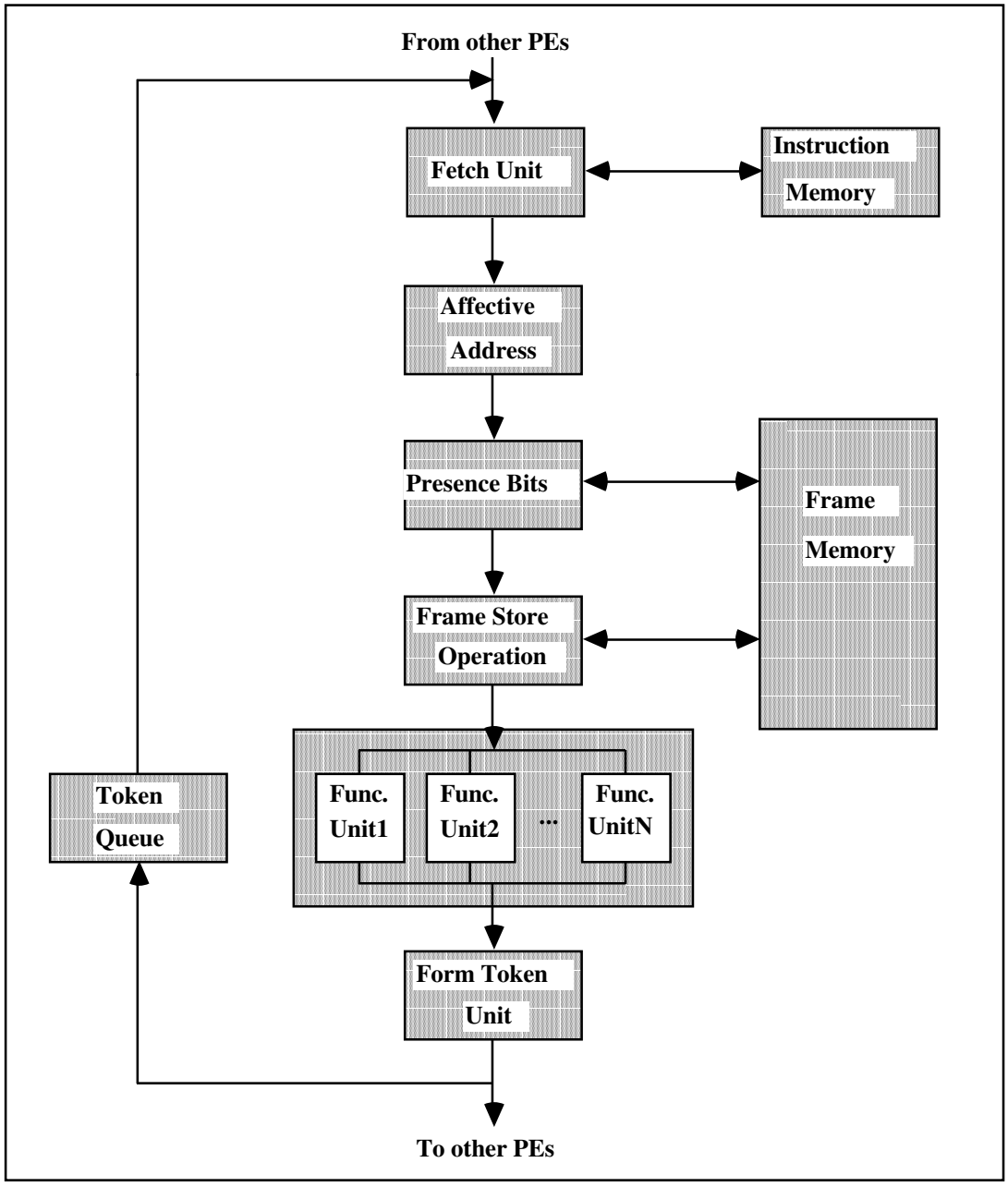

Figure 13. Structure of a typical explicit token-store dataflow PE

16.5 Hybrid multithreaded architectures

In the dataflow architectures instructions become self-scheduling based on the dataflow firing rule. Though this is a very useful mechanism for exploiting instruction level parallelism, it becomes superfluous for processing inherently sequential instruction streams and leads to unnecessary overhead in exploiting parallelism. In order to reduce overhead in dataflow machines an obvious compromise would be to increase granularity of parallelism and exploit parallelism only among procedures, meanwhile, inside procedures the sequential execution scheme of von Neumann processors could be applied. The resulting architecture is called the hybrid von Neumann/dataflow architecture which inherits features from both the dataflow and von Neumann processors.

Indeed von Neumann and dataflow processors represent the two extreme points of a range of architectures considering their instruction scheduling flexibility, latency toleration and synchronisation support. Table 1. shows these features for the three possible architecture types. Hybrid architectures can be classified according to their place at the scale of dataflow -> von Neumann architectures as it is depicted in Figure 14.

Table 1. Comparison of three architecture types

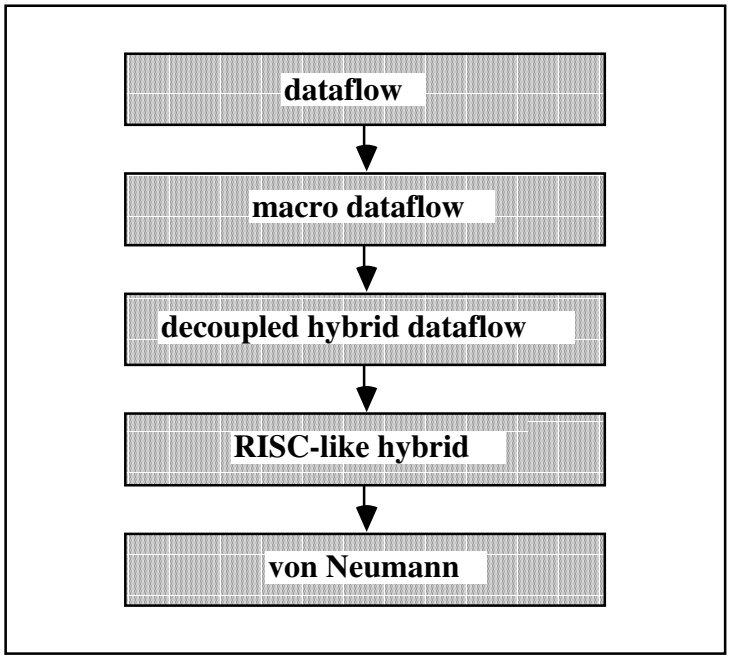

Figure 14. Scale of von Neumann/dataflow architectures

Each hybrid machine type is based on the concept of macro nodes applied in the dataflow graph. A certain part of the dataflow graph, where nodes cannot be or not worth being executed in parallel, is called a **macro node**. Instructions inside such a macro node are to be performed sequentially based on the conventional control flow semantics

represented by the program counter in von Neumann machines. However, macro nodes are executed in parallel, based on the data driven semantics, like any node in dataflow machines.

One can consider *macro nodes as threads and hybrid architectures as multithreaded processors*. The hybrid von Neumann/dataflow architectures represent the 4th generation of dataflow machines. Indeed, they all apply some form of the most developed dynamic, explicit token-store execution scheme for exploiting parallelism among the nodes of the dataflow graph.

16.5.1 Macro dataflow architectures

Macro dataflow architectures represent the first step from the conventional dataflow design towards hybrid multithreaded machines. They rather resemble to a dataflow architecture than to a von Neumann processor. Their typical structure is depicted in Figure 15. The main alteration to dataflow machines appears in the organisation of the internal control pipeline where instruction sequence is determined by a simple program counter like in von Neumann processors. Macro nodes of the dataflow graph are executed by this internal control pipeline. Parallelism among macro nodes and other nodes of the dataflow graph is exploited by the same execution mechanism that was applied in explicit token-store dataflow architectures.

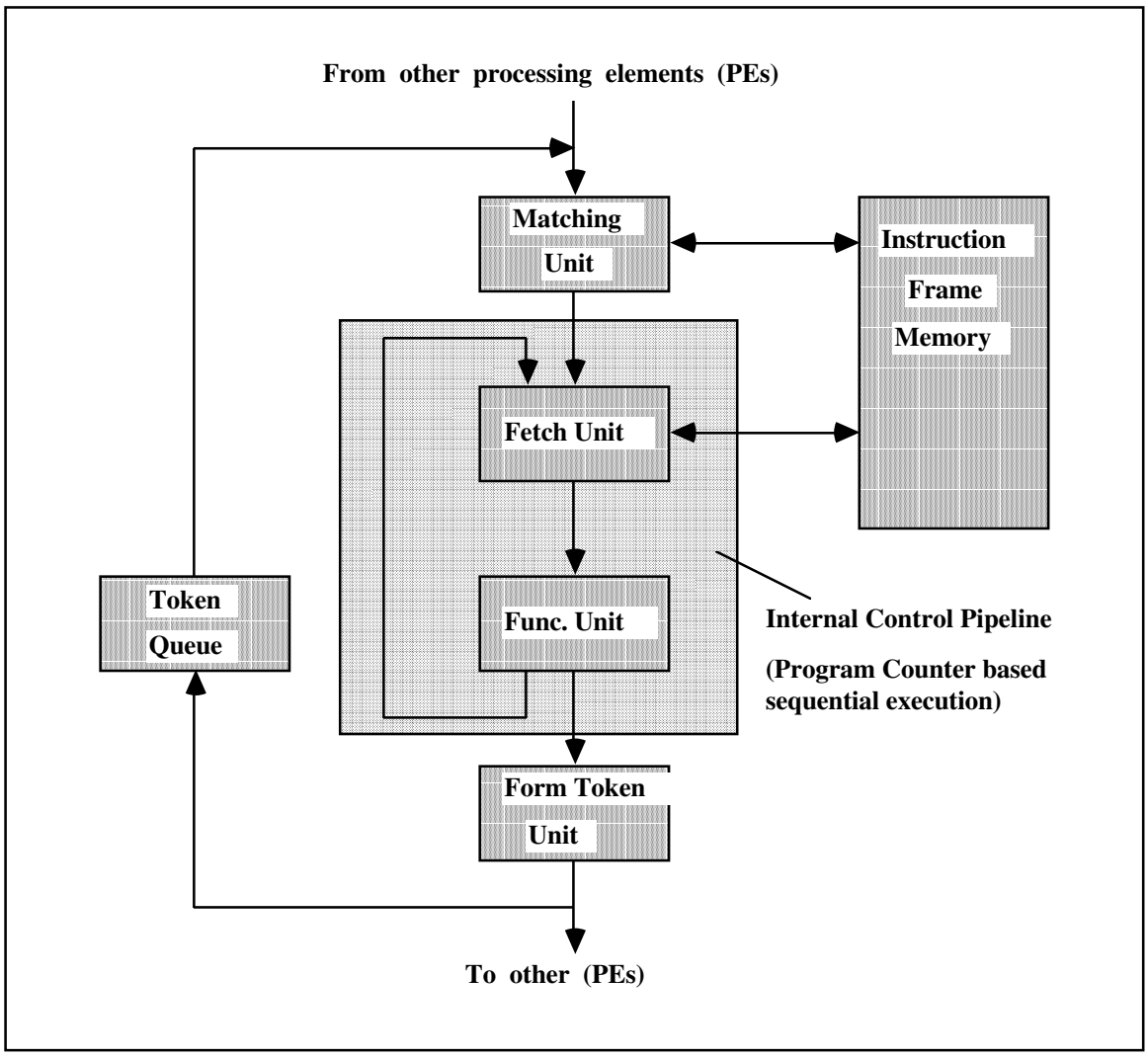

Figure 15. Structure of a typical macro dataflow PE

MIT Hybrid Machine

The MIT Hybrid Machine is a close follower of the Monsoon architecture. It introduces the concept of **scheduling quantum (SQ)**. An SQ represents a sequential partition in the dataflow graph composed according to any criterion of optimality. Ianucci defines five such criteria (Ianucci, 1988):

- maximisation of exploitable parallelism
- maximisation of run length in SQs
- maximisation of machine utilisation
- minimisation of explicit synchronisation
- deadlock avoidance.

Based on the criteria enumerated above the compiler can generate SQs in any dataflow graph. An SQ is implemented as a thread and represented as a continuation in the MIT Hybrid Machine. A continuation is a tuple consisting of a program counter (PC)

and frame base register (FBR). A continuation (or thread) can be in one of the following states:

Enabled: The continuation is placed in the Enabled Continuation Queue (which corresponds to the Token Queue of dataflow machines).
Running: The continuation is loaded into the Active Continuation Register.
Suspended: The continuation resides in a frame slot waiting for a matching token (similarly to the explicit token store machines).

A continuation remains in Running state as long as necessary input tokens are available and the last instruction is not reached yet. In the Running state the PC field of the continuation is updated exactly in the same way as in the von Neumann processors. Attempted read of an empty memory slot results in the suspension of the thread by saving its continuation into the empty memory slot and the presence bit of the memory slot is updated (see the I-structure in Section 16.4.2). The organisation of a processing element in the MIT Hybrid Machine is shown in Figure 16. Notice that optimised register use is possible in the control pipeline like in von Neumann processors. The I-structure memory is realised as a physically shared global memory in the MIT Hybrid Machine. The interconnection of processing elements and I-structure memory blocks is based on the "dance-hall" concept similarly to the Denelcor HEP machine.

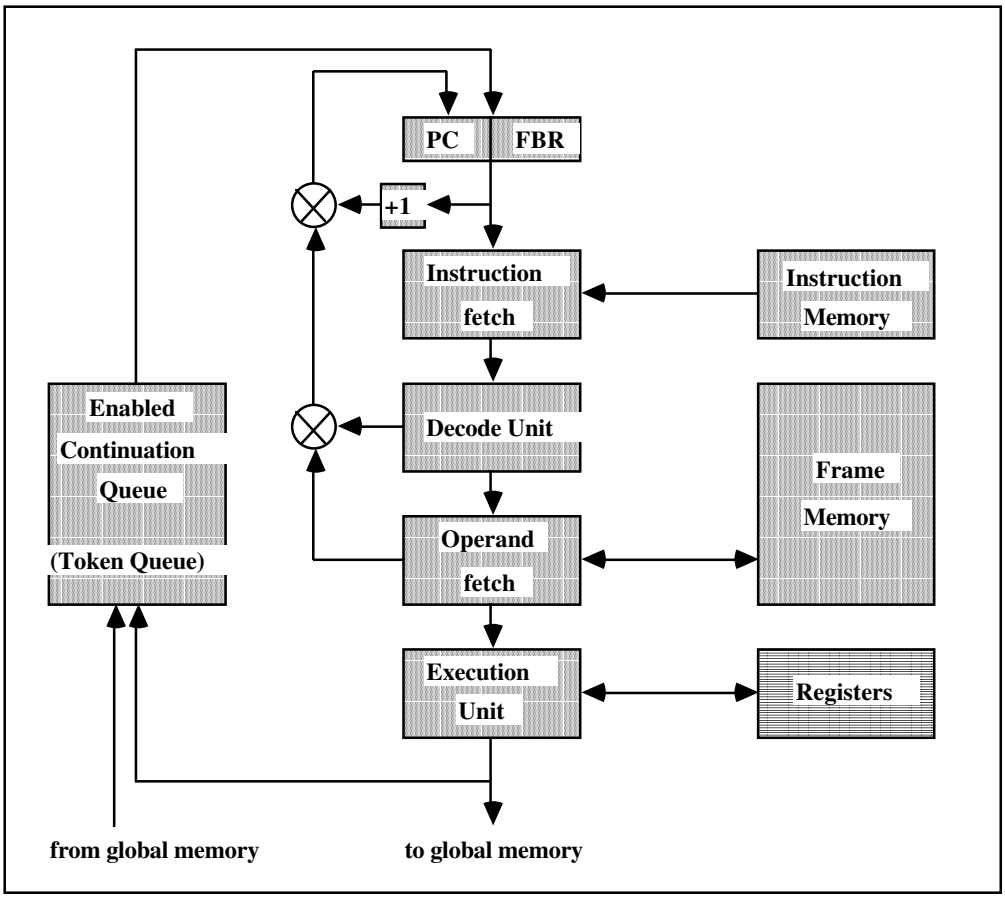

Figure 16. Organization of a PE in the MIT Hybrid Machine

The architecture of a processing element in the MIT Hybrid Machine is a real compromise between dataflow machines and von Neumann based parallel computers. On one hand it resembles to the Monsoon dataflow architecture, on the other hand it shows many similarities with von Neumann PEs like, for example, the Transputer (see Chapter 17). The Enabled Continuation Queue plays the same role as the two Active Process Queues in the Transputer. Communication and suspension on memory slots are very similar to the channel communication and suspension scheme of the Transputer since the channel is implemented as a two-state memory slot in the Transputer, too. The real difference is that the channel can transfer data of any complexity and hence after the synchronisation phase a physical communication phase is needed, meanwhile in the MIT Hybrid Machine a successful synchronisation results in the direct access of the required data.

Macro dataflow machines in Japan

The second example for the macro dataflow architecture class is the EM-4 machine developed at ETL of Japan. The EM-4 architecture is based on the **strongly connected arc model** where arcs of the dataflow graph are categorised as normal or strongly connected ones. Nodes inside a macro node - called a **strongly connected block (SCB)** - are connected by **strongly connected arcs** while SCBs are connected by normal arcs. An SCB can fire when all its input arcs contain a token, i.e. macro nodes are executed based on the dataflow semantics. However, nodes within a SCB are executed sequentially and therefore an SCB behaves as a thread (Sato et al.,1992).

The scheduling quantum (SQ) of the MIT Hybrid Machine and the strongly connected block (SCB) concepts are very similar since the execution of instructions in both kinds of macro nodes are sequential. The main difference between them occurs in the way they are activated. An SCB can be activated only when all of its input tokens are already available. Afterwards it can be executed without suspension. Execution of an SQ can be started when its first instruction (node) is enabled. If an instruction inside an SQ later encounters a missing data (token) the whole SQ (thread) execution is suspended on the empty memory slot. When the required data arrives the SQ becomes enabled again. The difference of the two concepts is illustrated by Figure 17 in which Figure 17.a shows a valid partition in the SQ concept. Both SQ1 and SQ2 can start their work when the values a, b and c are available. The only constraint is that I2 of SQ1 should wait for the result of I4 and similarly, I5 of SQ2 should wait for the completeness of I1. This mutual waiting would cause deadlock in case of SCB since none of the partitions could start their work until all the inputs are available, but they mutually would need input from the other partition. If the mutual dependence is extracted, as shown in Figure 17.b, the SCB

concept can be applied, too. First SCB2 can start its work and when I4 is completed, SCB1 becomes enabled, too.

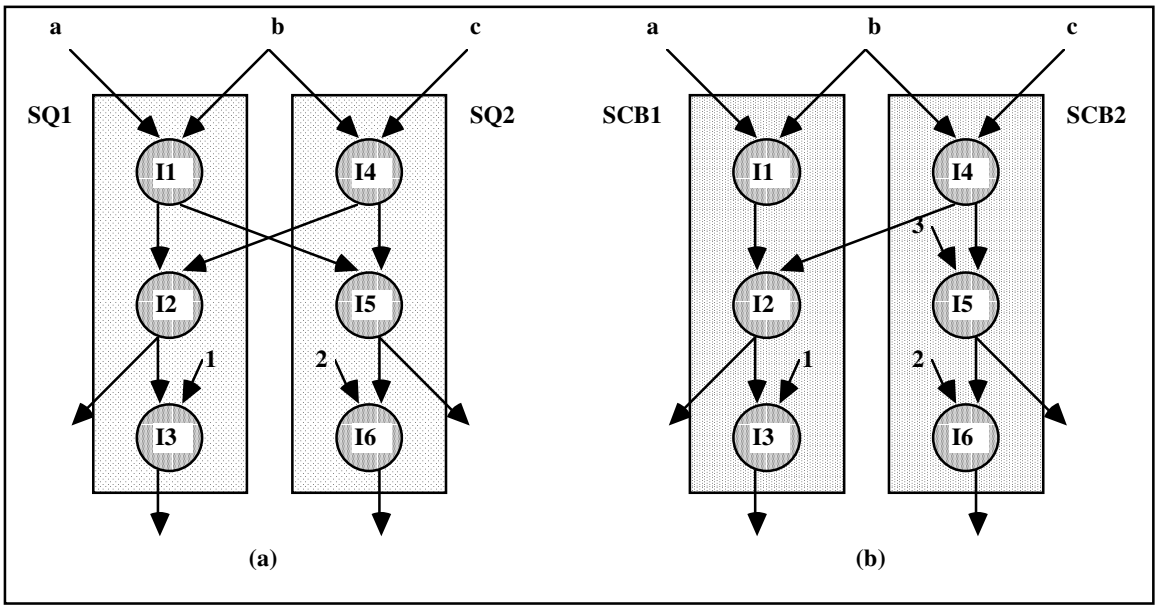

Figure 17. Comparison of SQ and SCB macro nodes

The third example for macro dataflow machines is the Datarol-II architecture that was developed at the Kyushu University of Japan (Kawano et al, 1995). The main advantage of the Datarol-II machine is the introduction of a hierarchical memory system including a preloading and an implicit load/store mechanism which results in a very low context switching overhead for threads.

16.5.2 Decoupled hybrid dataflow architectures

Decoupled hybrid dataflow architectures work according to the parallel control flow model based on control tokens (see Section 16.2.3). Since in this model instruction scheduling concerns the collection of necessary control tokens, a natural implementation of the model would require a separate specialised functional unit for this purpose. Indeed, decoupled hybrid dataflow architectures distinguish and separate a scheduling unit from the computational one. Two notable examples of this architecture type are:

1. USC Decoupled Architecture
2. McGill Dataflow Architecture and Super-Actor Machine (SAM)

The USC Decoupled Architecture

The USC Decoupled Architecture is based on the **Multilevel dataflow model** in which three kinds of nodes are distinguished (Evripidou and Gaudiot, 1991):

1. Scalar node (one node one operation)

2. Vector-macro-node
3. Compound macro node (CMN)

Vector-macro-nodes are specialised for vector operations. Their application makes the dataflow graph simpler and it also reduces the size of tags needed for vector and matrix operations. Compound macro nodes are collections of scalar and/or vector nodes. These are generated by the compiler like the SQ and SCB macro nodes of the macro dataflow architectures. The CMN is scheduled on the same principle as the SCB.

In the USC Decoupled Architecture the computation and scheduling are decoupled. Scheduling is realised by the Data-Flow Graph Engine (DFGE) which fetches enabled macro nodes (threads) from the Graph Memory. A macro node is enabled when all the input tokens have arrived to the node. This is administered by the status word of the node stored in the Graph Memory. The status word is originally set to the number of expected operands and decremented whenever an operand becomes ready. When it reaches zero, the node can fire. Computations within threads are coded in the Computation Memory and processed by the Computation Engine (CE) which is a von Neumann processor based on advanced pipeline techniques. The two engines communicate through two queues: Ready Queue (RQ) holds descriptor of enabled macro nodes, while Acknowledge Queue (AQ) contains descriptor of nodes executed by the CE. The DFGE updates the Graph Memory based on the control tokens found in the AQ. The structure of the USC Decoupled Architecture is shown in Figure 18.

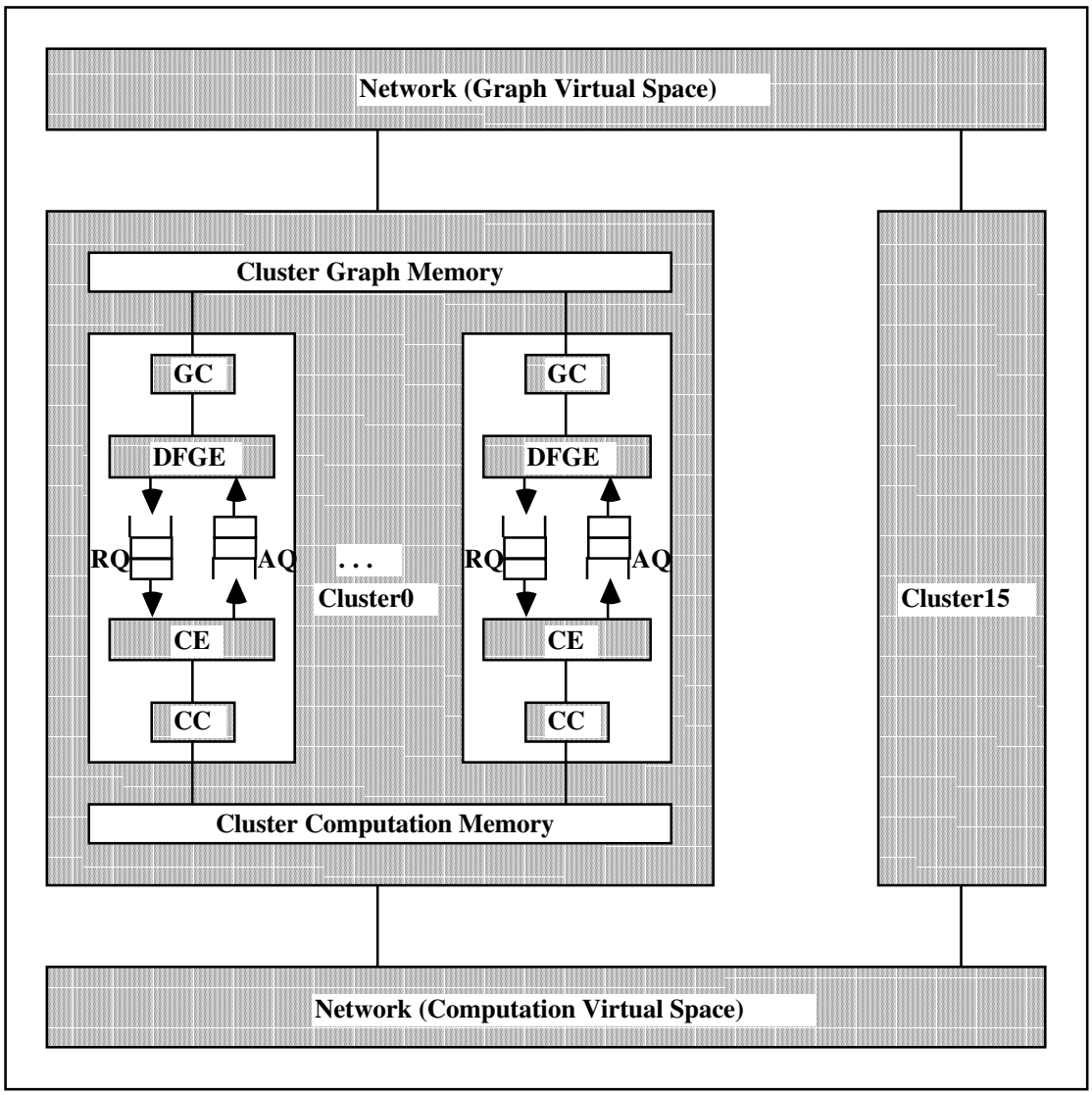

Figure 18. Structure of the USC Decoupled Architecture

The processing elements of the USC machine are clustered. There is a shared Graph and Computation Memory for each cluster. Cluster memories are connected via a communication network and constitute a global virtual shared memory. At each processing element two caches (GC = Graph Cache and CC = Computation Cache) assure fast local memory access. The placement and replacement policy of the caches is based on the contents of the Ready Queue. The aim is to have the context-blocks in the cache for all the instructions which are in the Ready Queue. If this condition can not be satisfied, the contents of the Ready Queue should be rearranged to give priority to the nodes whose context-blocks are already in the cache. Structure handling and producer/consumer synchronisation is based on a combination of the I-structure and token relabeling concepts.

The USC Decoupled Architecture has the following advantages:

1. The USC machine supports both thread-level and instruction-level parallelism based on compiler decision.

2. The Computation Engine can be any state-of-the-art RISC processor.

3. The number of threads per processor is not restricted by a register set. The only limit is the capacity of the Ready Queue.

The Super-Actor Machine (SAM)

The McGill Dataflow Architecture (MGDA) resembles to the USC Decoupled Architecture in many ways (Gao et al, 1989). The MGDA concept was developed towards a complete multithreaded machine, called the Super-Actor Machine (SAM). The novel feature of the SAM is the introduction of a special high-speed storage area called the **register-cache** (Hum and Gao, 1991). It is organised both as a register file viewed from the execution unit and as a fully associative cache memory from the main memory perspective.

In the SAM a thread is equivalent to a **super-actor** which is a macro node of the dataflow graph. Two types of super-actors are distinguished. Within a **sequential super-actor** the instructions must be executed sequentially due to data dependencies among them. Instructions in a **parallel super-actor** can be executed at every pipe beat since they are data independent. A super-actor should satisfy two conditions to become enabled:

1. **Firing rule** of static dataflow models: All the input arcs should contain a token and its output arcs should be empty (i.e. its result tokens generated from a previous activation are consumed).

2. **Space locality condition**: Its input data are physically residing in the register-cache and space is reserved there to store the results.

Due to the space locality condition once a super-actor becomes active it will be executed atomically until its completion without any suspension. The problem of remote load latency simply does not occur since every operation is entirely local to the register-cache. Synchronisation with other super-actors takes place in the Actor Scheduling Unit (ASU) based on the dataflow firing rule and hence a super-actor requires no synchronisation during its execution in the Super-actor Execution Unit (SEU).

The structure of the SAM is shown in Figure 19. The ASU processes the control tokens and decrements the associated enable count of actors. When an actor becomes enabled, the ASU resets its enable count and sends a fire signal containing the enabled actor along with its attributes to the Actor Preparation Unit (APU). The APU is a new unit in the SAM compared with the MGDA. Its main role is to allocate space in the register-cache for the enabled actors before passing them to the SEU. Notice that the

ASU implements the firing rule condition and the APU realises the space locality condition. By the time an actor arrives to the SEU it satisfies both conditions.

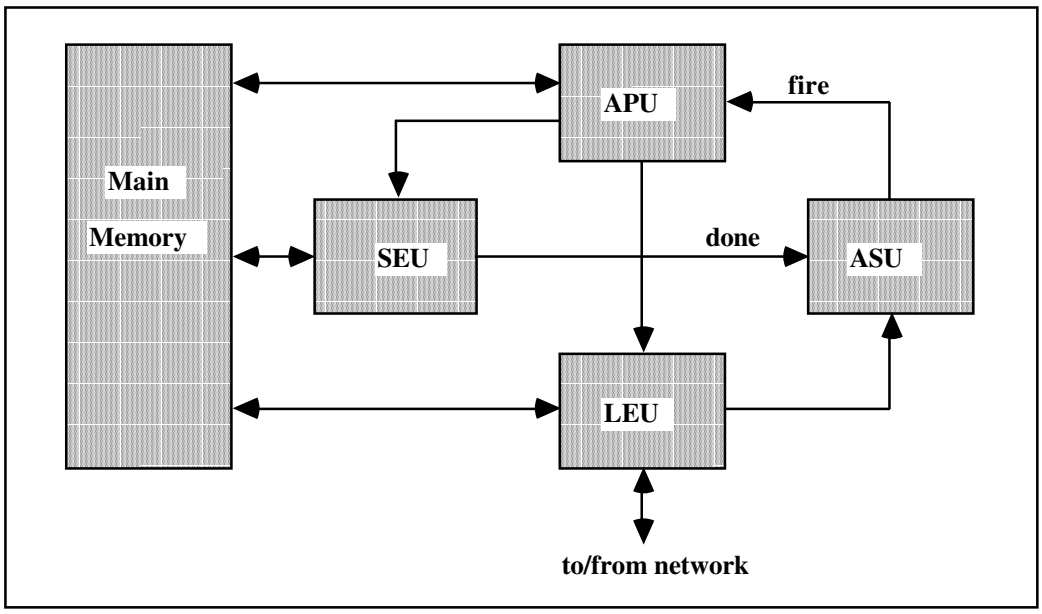

Figure 19. Structure of a node in the SAM

The SAM concept has the following advantages:

1. The SAM supports both thread-level and instruction-level parallelism based on compile-time decision.

2. The SEU is an optimised pipelined RISC processor enhanced with the register-cache.

3. The execution of a thread is atomic without any long-latency operation. By the time a thread becomes active, all the necessary data locations are placed in the local register-cache.

16.5.3 RISC-like hybrid multithreaded architectures

RISC-like hybrid multithreaded processors represent the closest architecture type to the von Neumann machines. Indeed, the starting point for designing these processors is a "conventional" RISC processor which is extended towards fine-grain thread-level parallelism support. Considering the computational model they are based on **parallel control operators** (**FORK** and **JOIN**) which represent the closest approach to the sequential control flow model. The main problem with this model, when it is applied at the process level as it was originally introduced, is that synchronisation on reusable memory cells is necessary. Moreover this synchronisation requires also significant overhead and special care when applied at the user level. The RISC-like hybrid

multithreaded architecture approach represents a successful attempt to solve all of these problems.

The main advantage of the hybrid multithreaded architectures is that they are able to solve the synchronisation in a fine-grain way with low overhead as a consequence of inheriting latency hiding techniques of dataflow machines. Advanced dataflow and control flow compiler techniques make it possible to generate the necessary control instructions (FORK and JOIN) from high level languages without introducing special user level notations.

There are two notable progress steps in designing hybrid multithreaded architectures:

1. the P-RISC processor developed at MIT and

2. the *T processor - a joint work of MIT and Motorola

In the next section a detailed description of these two hybrid multithreaded architecture designs are given.

P-RISC

P-RISC inherits the organisation of tokens from the macro dataflow machines, particularly from the MIT Hybrid Machine. Each token called as the **continuation** consists of two fields: frame pointer (FP) and instruction pointer (IP).

context = <FP.IP>

where the context represents a thread, FP represents the local data frame of the thread and IP points to the current instruction of the thread.

The structure of the P-RISC processor is shown in Figure 20. The pipeline of conventional RISC processors is easily recognisable in the figure. The main difference between P-RISC and RISC processors appears in the way how the next instruction is entered into the pipeline. In conventional RISC processors instructions from the same thread are loaded into the pipeline based on the program counter. In P-RISC the token queue contains threads and in each clock step a new thread enters the pipeline from the token queue. As a result different stages of the pipeline process on instructions belonging to different threads similarly to the HEP architecture. One can say that the P-RISC realises fine-grain thread-interleaving like the HEP. At the end of the processor pipeline P-RISC creates a new token and places it back to the token queue. Notice that the control loop based on the token queue and pipeline in the P-RISC is in principal the same as the Control Loop and pipeline in the HEP architecture.

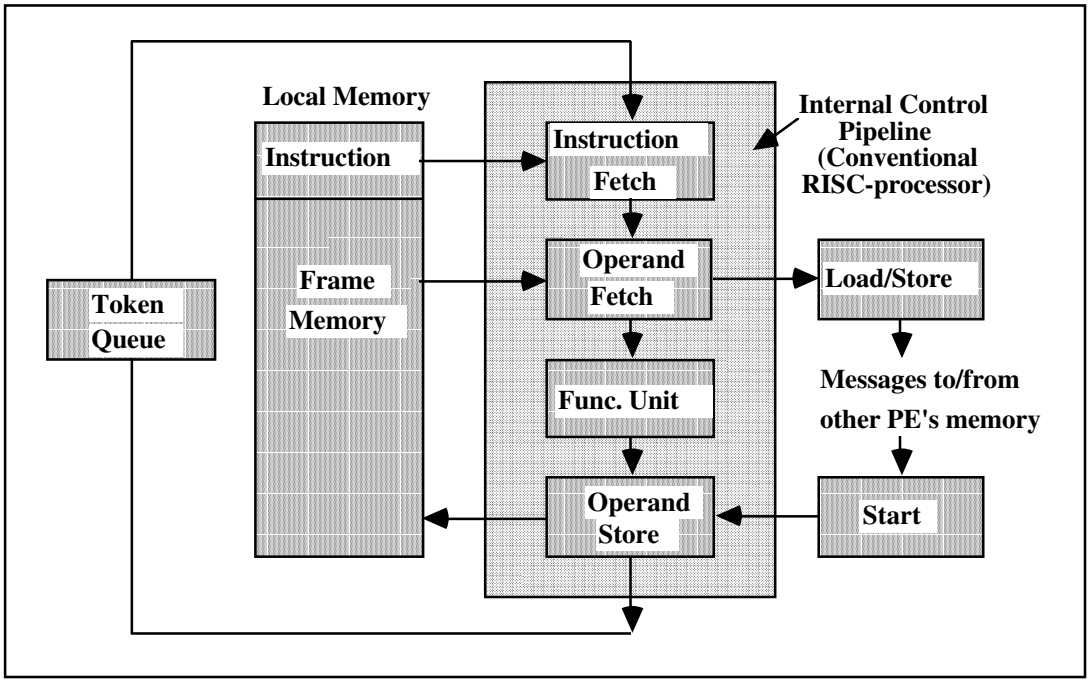

Figure 20. Structure of the P-RISC processing element

Considering ordinary RISC-like instructions, for most of them P-RISC works similarly to RISC processors. For example, an instruction in the form of

op s1 s2 d

is executed in the following way.

FP+d := [FP+s1] op [FP+s2]

i.e. the Frame Pointer (FP) serves as base address to fetch the source operands and to place the result of the instruction. Finally, a new token is created and placed back to the token queue:

<FP.IP+1>

Since conventional RISC processors can not tolerate memory latency when they execute load and store instructions, in the P-RISC design these instruction are implemented in an essentially different way. From a

load a x

instruction P-RISC creates a message in the form of

<READ, [FP+a], FP.IP+1, x>

which is sent to a remote memory module addressed by the content of the local memory location FP+a. It is the task of the remote processing element possessing the requested memory location to fetch the content (v) of the addressed memory cell and to send back as a reply message in the form of

<START, v, FP.IP+1, x>

to the requesting processor element. Notice that the requester processor does not create any local token after emitting the READ message and hence it does not place token in the local token queue. Rather, the updated continuation <FP.IP+1> of the thread issuing the load instruction temporarily travels in the interconnection network until the START message arrives back. At this time the requester processor stores the fetched value v into FP+x and places back the updated continuation <FP.IP+1> into the token queue.

The remote load operation realised this way has the following advantages:

1. Loads are split-phase transactions.

2. Any number of loads can be performed in the distributed system.

3. Responses can return in any order without disturbing the original semantics of the program.

4. The processor is not blocked during load operations (unlike conventional RISC processors).

5. Meanwhile a thread is travelling in the network other threads from the token queue can keep the processor busy. The only limit for the number of threads handled by one processor is the length of the token queue.

Notice that the Scheduling Function Unit (SFU) of the HEP is replaced by the split-phase transaction model of P-RISC. As a consequence the reissuing of load operations by the SFU is avoided in the P-RISC processor resulting in a much more efficient waiting mechanism for memory operations. Another advantage of P-RISC comparing to the HEP is that bubbles are not inserted into the processor pipeline during the remote memory load operation.

The store operation is also implemented by a message passing mechanism in the P-RISC based machine. The

store x a

instruction is executed by generating a

<WRITE, [FP+a], [FP+x]>

message directed to the owner PE of memory location addressed by the content of FP+a. However, in this case the updated continuation <FP.IP+1> of the thread can immediately be placed back to the token queue. Thread synchronisation on memory locations are realised by means of I-structure memories just like in the MIT Hybrid Machine.

In addition to the ordinary RISC-like instructions P-RISC implements new instructions to support fine-grain thread-level parallelism. These instructions used for creating new threads (fork) and removing threads (join) can be derived from the original dataflow graph of programs as it is shown in Figure 21. The effect of the

fork IPt

instruction is the insertion of two continuations into the token queue. The first one is the updated continuation of the current thread <FP.IP+1>, while the second one is a newly formed continuation <FP.IPt>. Afterwards the two threads represented by the two continuations can progress independently through the execution pipeline of P-RISC. When a node of the dataflow graph requires tokens from independent threads, a join instruction is generated by the compiler to collect the necessary tokens.

join x

represents a competition based reduction of threads in the machine. The caller thread checks the local memory cell FP+x. If it contains zero (empty), the current thread is removed from the system by not placing its continuation back to the token queue. At the same time FP+x is set to one (full). If FP+x contains full, the current thread can continue and its updated continuation <FP.IP+1> is inserted into the token queue. Notice that the technique is very similar to the explicit token-store (ETS) concept. The difference is that in the P-RISC control tokens, in the ETS data tokens are collected in the Frame Memory.

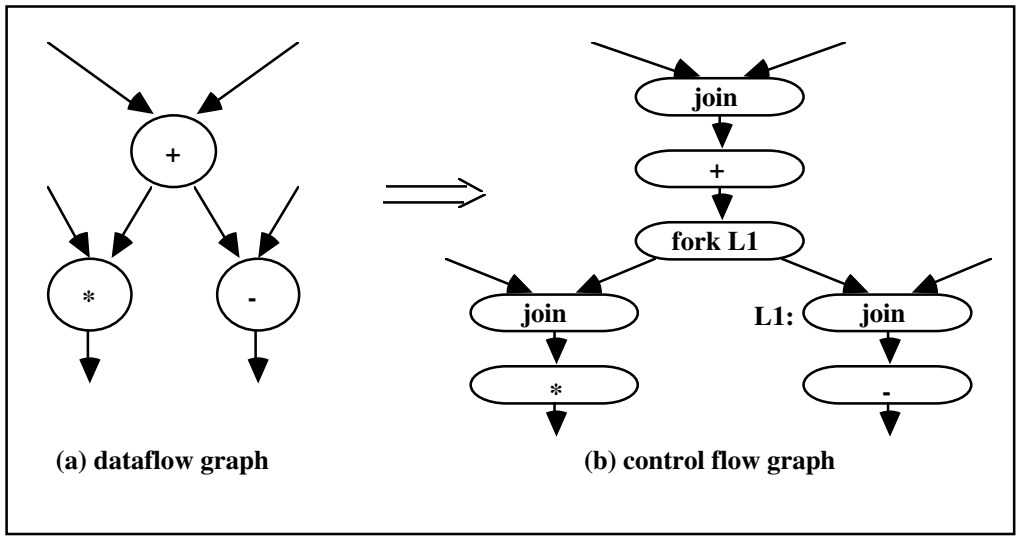

Figure 21. Transformation of dataflow graphs into control flow graphs

*T

The objective of the *T (read as start) design was to create a commercial multithreaded processor chip based on the principals of P-RISC. Careful analysis of the P-RISC model showed that the performance of the *T chip can significantly be improved by introducing functional coprocessors to support remote memory access. The structure of the *T chip is shown in Figure 22.

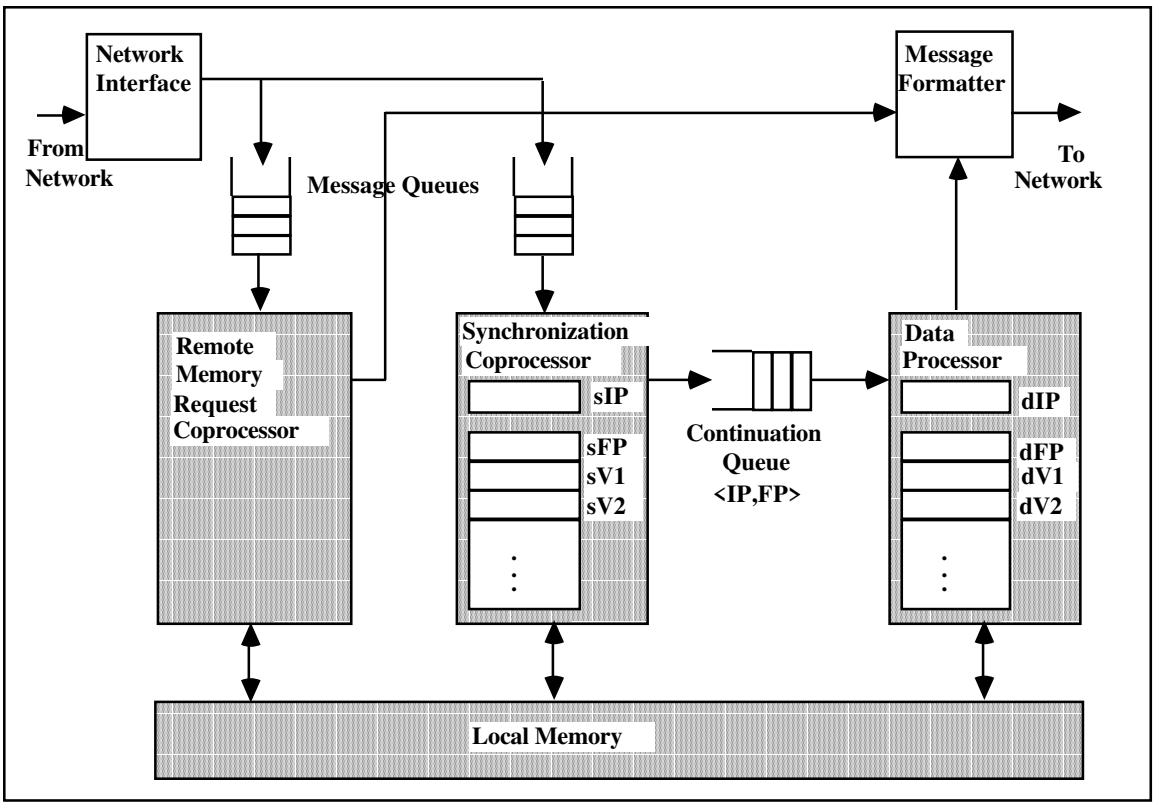

Figure 22. Structure of *T node (Nikhil & al. 1992)

The Data Processor contains the execution pipeline and operates on local instructions as it is shown in P-RISC. Whenever an instruction requiring remote memory access is fetched, it is passed to the Message Formatter which has the task to form a message from the instruction and send it into the network.

The Remote Memory Request Coprocessor handles incoming rload and rstore requests. It accesses the Local Memory to execute the requested services without disturbing the work of the Data Processor. In case of a rload instruction it fetches the information from the memory and passes it together with other parameters of the requesting message to the Message Formatter which composes a reply message and inserts it into the network.

The Synchronisation (SP) Coprocessor handles rload responses returning to this node:

<START, v, FP.IP+1, x>

SP loads FP+x with the received value v and places the updated continuation <FP.IP+1> into the Continuation Queue from which the Data Processor can resume the work on the thread.

16.6 Summary

The multithreaded architectures studied in this chapter are compared according to their design space in Table 2. Although the very first multithreaded machine was developed at the late seventies another decade was necessary to really understand the necessity of the multithreaded concept in the design of scalable parallel computers. Since 1988 the progress in the field of multithreading has been undiminished and year-by-year new multithreaded machines have been designed and even produced for commercial purposes.

Table 2. Comparison of multithreaded machines

Three main directions can be distinguished in the multithreaded machine concepts. The first direction is directly follows the ideas of the first multithreaded computer, the Denelcor HEP. The Tera machine is a representative example of this class of multithreaded machines. The main characteristics of this class are:

1. Von Neumann based computational model

2. Fine-grain thread interleaving

3. Large number of registers to accommodate as many threads as possible in the processor

The third feature makes this concept very expensive and depending on the current state of chip fabrication technology. This is an obvious drawback which will lead to the search of new forms of multithreading.

The second direction is represented by the MIT Alewife machine and has the following main properties:

1. Von Neumann based computational model

2. Coarse-grain thread interleaving

3. Small number of threads/processor

4. Coherent cache support to reduce the frequency of thread switches

Notice that this is the only scheme which combines multithreading with coherent cache memory in the framework of a CC-NUMA architecture. Since CC-NUMA machines are expected to become very popular as scalable parallel computers, it can be foreseen that the results of the MIT Alewife project will influent the design of future CC-NUMA architectures.

Interestingly the third direction has grown from the field of dataflow architectures and in the form of the *T processor it has led to a concept very similar to the initial Denelcor HEP. Within this direction two significantly different concepts can be distinguished:

a. Hybrid von Neumann/dataflow architectures applying coarse-grain thread interleaving

b. Hybrid von Neumann/dataflow architectures supporting fine-grain thread interleaving

Followers of the first concept apply some form of macro nodes in the dataflow graph. They represent a real hybrid design. From the perspective of macro node communication and synchronisation they look like a dataflow machine. Concerning inside execution scheme of macro nodes they behave as von Neumann processors. All of them require sophisticated compiler technology to optimally partitioning the dataflow graph into macro nodes.

The second hybrid concept gets very close to the original HEP design although significantly improves its disadvantageous features. By introducing the concepts of continuation and control pipeline based on the continuation queue radically reduces the number of necessary registers required for fast context switching. The split-phase memory access scheme avoids the busy-wait memory access mechanism of the HEP. Thread creation is straightforward by systematically transforming the dataflow graph into the control flow graph based on FORK and JOIN instructions. The recent *T design represents a very promising approach to efficiently support latency hiding by multithreading.