PARALLEL SYSTEMS LABORATORY
Department of Electrical Engineering

ניסוי מבוא למיקרו-מעבד מרובה ליבות

מהדורת ניסיון – עדכון אחרון: דצמבר, 2013

נכתב ע"י: פבל ליפשיץ, עוז שמואלי

מהדורה קודמת: עמית ברמן

הטכניון – מכון טכנולוגי לישראל
הפקולטה להנדסת חשמל

Table of Contents

רשימת איורים

כללי בטיחות במעבדה למערכות מקביליות

קרא בעיון והקפד על שמירת הכללים !

1. אין לעבוד במעבדה ללא קורס בטיחות במעבדות חשמל (044102).
2. חובה על הסטודנטים לקיים את הוראות הבטיחות כפי שנלמדו בקורס הבטיחות.
3. אין לקיים פעילות במעבדה ביחידות.
4. אין לבצע חיבור או ניתוק של מכשירי חשמל (לרבות של מכשירים המתחברים למחשב). אם נדרשת פעולה כזו היא מחייבת נוכחות של אחד מחברי צוות המעבדה.
5. במידה וקיימת תקלה כלשהי במערכת הניסוי ו/או המחשב **אין לנסות לתקנה לבד**, אלא לקרוא לאחד מחברי צוות המעבדה.
6. בשום מקרה **אסור לפרק את המחשב או מערכת הניסוי !**

בכל מקרה – חובה לדווח על כל ליקוי בטיחות למען ביטחונך ושלומם של אחרים.

טלפונים בחרום	מרפאה: 2545, 3037, בטחון: 2475 , מגן דוד אדום: 101 מכבי אש:102
כיבוי אש	מטף נמצא בכל מעבדה ליד דלת הכניסה למעבדה. זרנוק כיבוי נמצא ליד יציאת חרום בקצה הפרוזדור
מפסק חשמל ראשי	מפסק החשמל נמצא בתוך המעבדה, בארון החשמל ליד דלת היציאה.
יציאות חרום	דלת ליציאת חרום נמצאת בקצה המסדרון.

ביצוע הניסוי במעבדה מותנה בקריאת ההוראות הנ"ל, ועמידה בבוחן הבטיחות הפקולטי.

אין לגשת לביצוע הניסוי ללא קיום התנאים הנ"ל!

הנחיות כלליות לניסוי

מטרת הניסוי:

הקניית ידע בסיסי במערכות עיבוד מקבילי, מבני מעבדים ואפשרויות ההאצה כתוצאה מעיבוד מקבילי. יינתן דגש על התאמת אלגוריתם לביצוע מקבילי ושיקולי עלות/תועלת ברמות המיקבול השונות.

בניסוי שני חלקים והוא מבוצע בשני מפגשים של 4 שעות כל אחד. לקראת כל מפגש נדרשת עבודת הכנה, ובסיום הניסוי יש לכתוב ולהגיש דו"ח מסכם

חלק א': היכרות עם סביבת הפיתוח, מבנה מעבד ה-HAL ואופן כתיבת קוד ב- Task Level Parallelism.

חלק ב': שילוב

קדם לניסוי: תכן לוגי (044262) (רצוי: מבנה מערכות תכנה ו/או מבנה מחשבים ו/או ידע רלוונטי בתחום).

סביבת עבודה: עמדת PC מקושרת ברשת מקומית עם מערכת הפעלה Windows 7 32-bit.

הניסוי נערך בסביבת הפיתוח Eclipse על בסיס Simulator של מעבד HAL (HyperCore Architecture) של חברת Plurality.

כתיבת התכנה בשפת C.

בשלב זה, הניסוי מתבצע בחשבון מחשב מקומי. לכן יש לשמור את הקבצים בחשבון אישי אחר (למשל באמצעות FTP) או בעזרת התקני USB. אין אחריות לקבצים שיושארו בחשבון המקומי. הכניסה לחשבון המקומי תעשה ע"י מנחה הניסוי בלבד.

מיקום: המעבדה למערכות מקביליות, בניין הנדסת חשמל (מאייר), קומה 12 , חדר 1237.

אחראי: עוז שמואלי, חדר 1242 , טל. 4691 shmueli@ee.technion.ac.il

הנחיות לביצוע הניסוי:

- לכל פגישה יש להגיש דו"ח הכנה מלא.
- לפני כל פגישה יש לעבור בבית על מהלך הניסוי כדי להיות מוכנים ובקיאים במהלך הניסוי.

הגשת דו"ח:

יכלול, עבור כל סעיף שבוצע:

- תיעוד והסבר לגבי כל תוצר של הניסוי: התוכנית שנכתבה, רמות מיקבול ו תלויות.
- תשובות לכל הסעיפים עם הסברים וגרפים ע"פ הצורך.
- הגשה - הכנה: לפני כל פגישה. מסכם: לא יאוחר משבועיים מתום הניסוי.

את הדו"חות יש להגיש <u>באופן אלקטרוני בלבד</u>, במערכת LabAdmin.

זיכרו לציין בדו"ח את השמות, מספרי הסטודנט, מספרי טלפון וכתובות דוא"ל!

ציון כל פגישה ו[illegible]רכב באופן הבא:

- ציון על דו"ח ההכנה (25%).
- התרשמות המדריך ממהלך הניסוי במעבדה (40%).
- התרשמות המדריך ממוכנות הסטודנטים לניסוי, הבנת החומר וכן הערכה כללית (10%).
- ציון על דו"ח מסכם (25%).
- במהלך הניסוי תיערך בחינה קצרה בע"פ על ההכנה, מהלך הניסוי ותוצאותיו.

בהצלחה !

Part A – Parallel computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.

Traditionally, computer software has been written for serial computation, in a way that enabled execution of one instruction at a time, and after instruction is finished the next can be executed.

Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above.

Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. An increase in frequency thus decreases runtime for all compute-bound programs.

However, power consumption by a chip is given by the equation:

$$\boxed{P = C \cdot V^2 \cdot f}$$

Where P is power, C is the capacitance being switched per clock cycle (proportional to the number of transistors whose inputs change), V is voltage, and f is the processor frequency (cycles per second).

Moore's Law is the empirical observation that transistor density in a microprocessor doubles every 18 to 24 months. With the end of frequency scaling, these additional transistors (which are no longer used for frequency scaling) can be used to add extra hardware for parallel computing.

Amdahl's law

Optimally, the speed-up from parallelization would be linear—doubling the number of processing elements should have the runtime, and doubling it a second time should again have the runtime. However, very few parallel algorithms achieve optimal speed-up. Most of them have a near-linear speed-up for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

The potential speed-up of an algorithm on a parallel computing platform is given by Amdahl's law, originally formulated by Gene Amdahl in the 1960s. It states that a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization. A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. If α is the fraction of running time a program spends on non-parallelizable parts, then:

$$\boxed{\lim_{P\to\infty} \frac{1}{\frac{1-\alpha}{P}+\alpha} = \frac{1}{\alpha}}$$

is the maximum speed-up with parallelization of the program. If the sequential portion of a program accounts for 10% of the runtime ($\alpha = 0.1$), we can get no more than a 10× speed-up, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. "When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned."

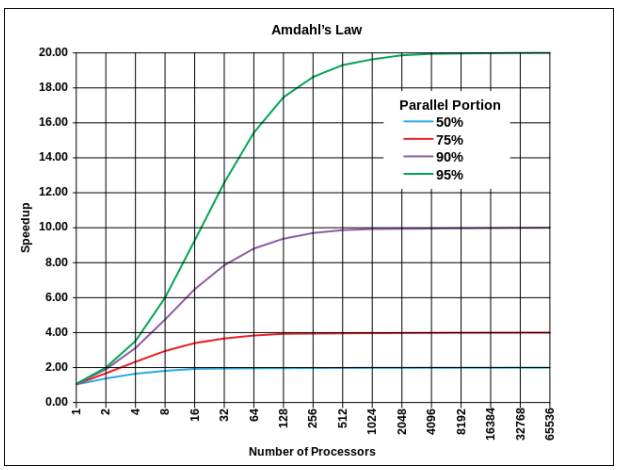


Figure 1 - Amdahl's Law

Figure 1 illustrates the consequences from Amdahl's Law: the speed-up of a program from parallelization is limited by how much of the program can be parallelized. For example, if 90% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 10x no matter how many processors are used.

Amdahl's law assumes that the entire problem is of fixed size so that the total amount of work to be done in parallel is also *independent of the number of processors.* Gustafson's law is another law in computing which closely related to Amdahl's law, but assumes that the total amount of work to be done in parallel varies linearly with the number of processors.

Dependencies

Understanding data dependencies is fundamental in implementing parallel algorithms. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.

Let P_i and P_j be two program segments. Bernstein's conditions describe when the two are independent and can be executed in parallel. For P_i, let I_i be all of the input variables and O_i the output variables, and likewise for P_j. P_i and P_j are independent if they satisfy

$$I_j \cap O_i = \varnothing$$
$$I_i \cap O_j = \varnothing$$
$$O_j \cap O_j = \varnothing$$

Violation of the first condition introduces a flow dependency, corresponding to the first segment producing a result used by the second segment. The second condition represents an anti-dependency, when the second segment (P_j) produces a variable needed by the first segment (P_i). The third and final condition represents an output dependency: When two segments write to the same location, the result comes from the logically last executed segment.

Consider the following functions, which demonstrate several kinds of dependencies:

```
1: function Dep(a, b)
2: c := a·b
3: d := 3·c
4: end function
```

Operation 3 in Dep(a, b) cannot be executed before (or even in parallel with) operation 2, because operation 3 uses a result from operation 2. It violates condition 1, and thus introduces a flow dependency.

```
1: function NoDep(a, b)
2: c := a·b
3: d := 3·b
4: e := a+b
5: end function
```

In this example, there are no dependencies between the instructions, so they can all be run in parallel.

Bernstein's conditions do not allow memory to be shared between different processes. For that, some means of enforcing an ordering between accesses is necessary, such as semaphores, barriers or some other synchronization method.

Race conditions, mutual exclusion, synchronization, and parallel slowdown

Subtasks in a parallel program are often called threads. Some parallel computer architectures use smaller, lightweight versions of threads known as fibers, while others use bigger versions known as processes. Threads will often need to update some variable that is shared between them. The instructions between the two programs may be interleaved in any order. For example, consider the following program:

Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A: Write back to variable V	3B: Write back to variable V

If instruction 1B is executed between 1A and 3A, or if instruction 1A is executed between 1B and 3B, the program will produce incorrect data. This is known as a race condition. The programmer must use a lock to provide mutual exclusion. A lock is a programming language construct that allows one thread to take control of a variable and prevent other threads from reading or writing it, until that variable is unlocked. The thread holding the lock is free to execute its critical section (the section of a program that requires exclusive access to some variable), and to unlock the data when it is finished. Therefore, to guarantee correct program execution, the above program can be rewritten to use locks:

Thread A	Thread B
1A: Lock variable V	1B: Lock variable V
2A: Read variable V	2B: Read variable V
3A: Add 1 to variable V	3B: Add 1 to variable V
4A: Write back to variable V	4B: Write back to variable V
5A: Unlock variable V	5B: Unlock variable V

One thread will successfully lock variable V, while the other thread will be locked out—unable to proceed until V is unlocked again. This guarantees correct execution of the program. Locks, while necessary to ensure correct program execution, can greatly slow a program.

Many parallel programs require that their subtasks act in synchrony. This requires the use of a barrier. Barriers are typically implemented using a software lock. One class of algorithms, known as lock-free and wait-free algorithms, altogether avoids the use of locks and barriers. However, this approach is generally difficult to implement and requires correctly designed data structures.

Not all parallelization results in speed-up. Generally, as a task is split up into more and more threads, those threads spend an ever-increasing portion of their time communicating with each other. Eventually, the overhead from communication dominates the time spent solving the problem, and further parallelization (that is, splitting the workload over even more threads) increases rather than decreases the amount of time required to finish. This is known as parallel slowdown.

Fine-grained, coarse-grained, and embarrassing parallelism

Applications are often classified according to how often their subtasks need to synchronize or communicate with each other. An application exhibits fine-grained parallelism if its subtasks must communicate many times per second; it exhibits coarse-grained parallelism if they do not communicate many times per second, and it is embarrassingly parallel if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize.

Consistency models

Parallel programming languages and parallel computers must have a consistency model (also known as a memory model – we will discuss this in details in the following topics). The consistency model defines rules for how operations on computer memory occur and how results are produced.

Flynn's taxonomy

Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy.

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

The single-instruction-single-data (SISD) classification is equivalent to an entirely sequential program. The single-instruction-multiple-data (SIMD) classification is analogous to doing the same operation repeatedly over a large data set. This is commonly done in signal processing applications. Multiple-instruction-single-data (MISD) is a rarely used classification. While computer architectures to deal with this were devised (such as systolic arrays), few applications that fit this class materialized. Multiple-instruction-multiple-data (MIMD) programs are by far the most common type of parallel programs.

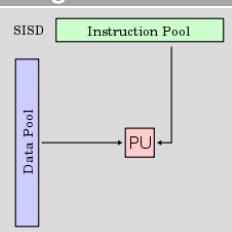
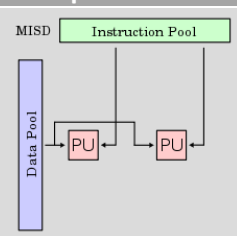
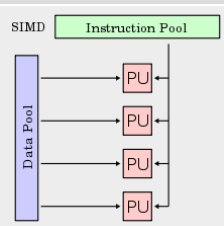
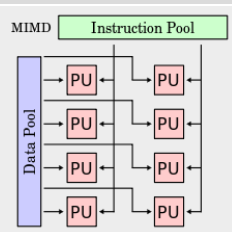
	Single instruction	Multiple instruction
Single data		
Multiple data		

Figure 2 - Flynn's taxonomy

SISD is referring to a computer architecture in which a single processor, a uniprocessor, executes a single instruction stream, to operate on data stored in a single memory. This corresponds to the von Neumann architecture. SISD can have concurrent processing characteristics. Instruction fetching and pipelined execution of instructions are common examples found in most modern SISD computers.

MISD (multiple instruction, single data) is referring to a type of parallel computing architecture where many functional units perform different operations on the same data. Pipeline architectures belong to this type. Not many instances of this architecture exist, but one prominent example of MISD in computing are the Space Shuttle flight control computers.

SIMD describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Thus, such machines exploit data level parallelism. SIMD is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio. Most modern CPU designs include SIMD instructions in order to improve the performance of multimedia use. An application that may take advantage of SIMD is one where the same value is being added to a large number of data points, a common operation in many multimedia applications. Figure 4 illustrates the benefits of using SIMD instruction: in ordinary tripling of four 8-bit numbers. The CPU loads one 8-bit number into R1, multiplies it with R2, and then saves the answer from R3 back to RAM. This process is repeated for each number, whereas in SIMD tripling of four 8-bit numbers, the CPU loads 4 numbers at once, multiplies them all in one SIMD-multiplication, and saves them all at once back to RAM. In theory, the speed up is about 75%.

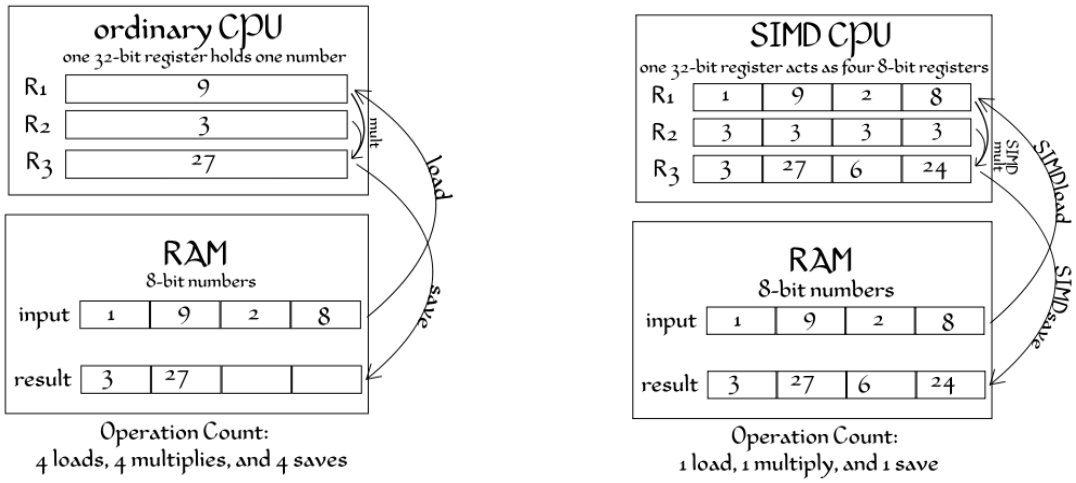


Figure 3 - Oridnary CPU vs. SIMD CPU

Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data. MIMD machines can be of either shared memory or distributed memory categories.

In Shared Memory Model the processors are all connected to a "globally available" memory, via either a software or hardware means. The operating system usually maintains its memory coherence. From a programmer's point-of-view, this memory model is better understood than the distributed memory model. Another advantage is that memory coherence is managed by the operating system and not the written program. Two known disadvantages are: scalability beyond thirty-two processors is difficult, and

the shared memory model is less flexible than the distributed memory model. In the next section we will see how the HAL processor addressed this issues[. Shared memory access can be distinguished to two access types: Bus-based in which all processors are attached to a bus which connects them to memory. Means that every machines with shared memory share a specific CM. Machines with hierarchical shared memory use a hierarchy of buses to give processors access to each other's memory. Processors on different boards may communicate through inter-nodal buses. Buses support communication between boards. With this type of architecture, the machine may support over a thousand processors.

In distributed memory MIMD machines, each processor has its own individual memory location. Each processor has no direct knowledge about other processor's memory. For data to be shared, it must be passed from one processor to another as a message. Since there is no shared memory, contention is not as great a problem with these machines. It is not economically feasible to connect a large number of processors directly to each other. A way to avoid this multitude of direct connections is to connect each processor to just a few others. This type of design can be inefficient because of the added time required to pass a message from one processor to another along the message path. The amount of time required for processors to perform simple message routing can be substantial.

Types of parallelism

Bit-level parallelism

From the advent of very-large-scale integration (VLSI) computer-chip fabrication technology in the 1970s until about 1986, speed-up in computer architecture was driven by doubling computer word size—the amount of information the processor can manipulate per cycle. Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose sizes are greater than the length of the word. For example, an 8-bit processor requires two instructions to complete a single operation of adding two 16-bit integers, where a 16-bit processor would be able to complete the operation with a single instruction.

Historically, 4-bit microprocessors were replaced with 8-bit, then 16-bit, then 32-bit microprocessors. This trend generally came to an end with the introduction of 32-bit processors, which has been a standard in general-purpose computing for two decades. Not until recently with the advent of x86-64 architectures, have 64-bit processors become commonplace.[if3]

Instruction-level parallelism

A computer program, is in essence, a stream of instructions executed by a processor. These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism. Advances in instruction-level parallelism dominated computer architecture from the mid-1980s until the mid-1990s.

Modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage; a processor with an N-stage pipeline can have up to N different instructions at different stages of completion. The canonical example of a pipelined processor is a RISC processor, with five stages: instruction fetch, decode, execute, memory access, and write back. The Pentium 4 processor had a 35-stage pipeline.

In addition to instruction-level parallelism from pipelining, some processors can issue more than one instruction at a time. These are known as superscalar processors. Instructions can be grouped together only if there is no data dependency between them.

Task parallelism

Task parallelism is the characteristic of a parallel program that "entirely different calculations can be performed on either the same or different sets of data"[14]. This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data. Task parallelism does not usually scale with the size of a problem.

Figure 5 illustrates the different levels of parallelism

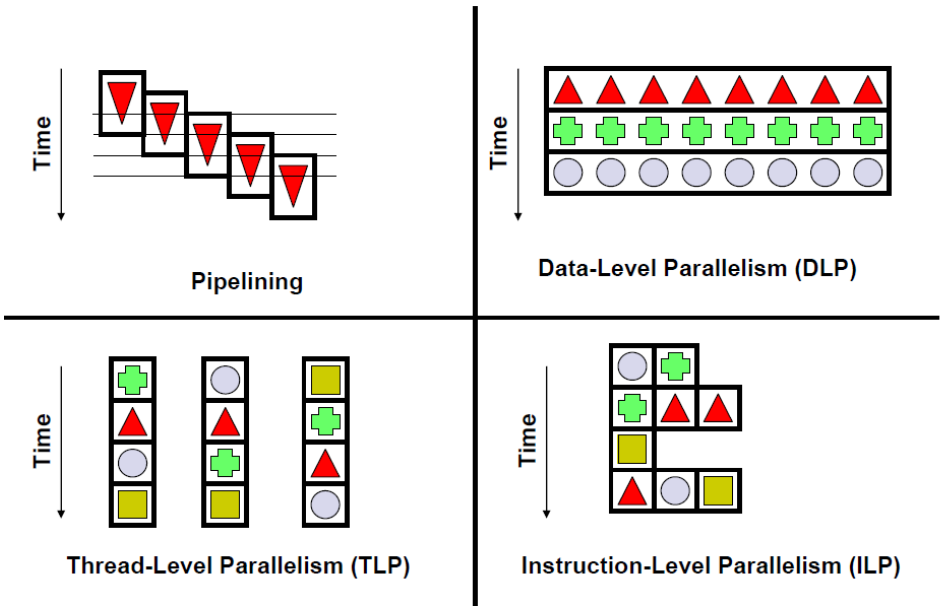


Figure 4 - Different levels of parallelism

Combination of levels of parallelism

Current computers allow exploiting of many parallel modes at the same time for maximum combined effect. A distributed memory program may run on a collection of nodes. Each node may be a shared memory computer and execute in parallel on multiple CPUs. Within each CPU, SIMD vector instructions (usually generated automatically by the compiler) and superscalar instruction execution (usually handled transparently by the CPU itself), such as pipelining and the use of multiple parallel functional units, are used for maximum single CPU speed.

Classes of parallel computers

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes. These are not mutually exclusive; for example, clusters of symmetric multiprocessors are relatively common.

Multicore computing

A multicore processor is a processor that includes multiple execution units ("cores") on the same chip. These processors differ from superscalar processors, which can issue multiple instructions per cycle from one instruction stream (thread); in contrast, a multicore processor can issue multiple instructions per cycle from multiple instruction streams.

Each core in a multicore processor can potentially be superscalar as well—that is, on every cycle, each core can issue multiple instructions from one instruction stream. A processor capable of simultaneous multithreading has only one execution unit ("core"), but when that execution unit is idling (such as during a cache miss), it uses that execution unit to process a second thread.

Symmetric multiprocessing

A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus. Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors. "Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists."

Distributed computing

A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable.

Cluster computing

A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. While machines in a cluster do not have to be symmetric, load balancing is more difficult if they are not.

Massive parallel processing

A massively parallel processor (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks (whereas clusters use commodity hardware for networking). MPPs also tend to be larger than clusters, typically having "far more" than 100 processors. In a MPP, "each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect."

Grid computing

Distributed computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the Internet, distributed computing typically deals only with embarrassingly parallel problems.

Most grid computing applications use middleware, software that sits between the operating system and the application to manage network resources and standardize the software interface. Often, distributed

computing software makes use of "spare cycles", performing computations at times when a computer is idling.

General-purpose computing on graphics processing units (GPGPU)

General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in computer engineering research. GPUs are co-processors that have been heavily optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations—particularly linear algebra matrix operations. In the early days, GPGPU programs used the normal graphics APIs for executing programs. However, several new programming languages and platforms have been built to do general purpose computation on GPUs with both Nvidia and AMD releasing programming environments with CUDA and Stream SDK respectively.

Vector processors

A vector processor is a CPU or computer system that can execute the same instruction on large sets of data. "Vector processors have high-level operations that work on linear arrays of numbers or vectors. An example vector operation is A = B × C, where A, B, and C are each 64-element vectors of 64-bit floating-point numbers." They are closely related to Flynn's SIMD classification.

Part B – Hardware

Memory and communication

Main memory in a parallel computer is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space). Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well. Distributed shared memory and memory virtualization combine the two approaches, where the processing element has its own local memory and access to the memory on non-local processors. Accesses to local memory are typically faster than accesses to non-local memory.

Computer architectures in which each element of main memory can be accessed with equal latency and bandwidth are known as Uniform Memory Access (UMA) systems. Typically, that can be achieved only by a shared memory system, in which the memory is not physically distributed. A system that does not have this property is known as a Non-Uniform Memory Access (NUMA) architecture. Distributed memory systems have non-uniform memory access.

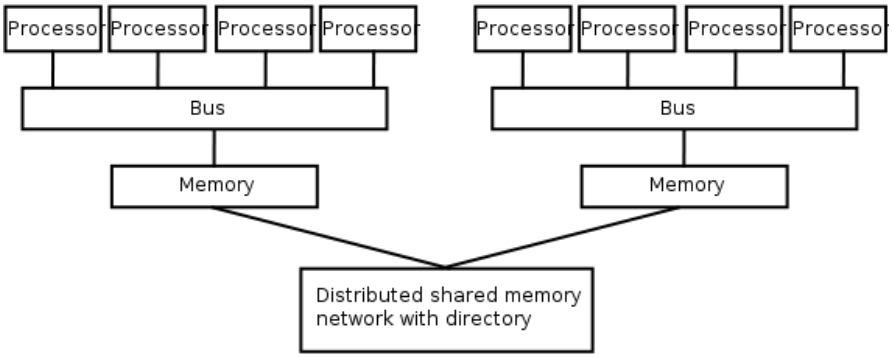


Figure 5 - A logical view of a Non-Uniform Memory architecture

Figure 6 illustrates the fact that processors in one directory can access that directory's memory with less latency than they can access memory in the other directory's memory.

Computer systems make use of caches—small, fast memories located close to the processor which store temporary copies of memory values (nearby in both the physical and logical sense). Parallel computer systems have difficulties with caches that may store the same value in more than one location, with the possibility of incorrect program execution. These computers require a cache coherency system, which keeps track of cached values and strategically purges them, thus ensuring correct program execution. Designing large, high-performance cache coherence systems is a very difficult problem in computer architecture. As a result, shared-memory computer architectures do not scale as well as distributed memory systems do.

Processor–processor and processor–memory communication can be implemented in hardware in several ways, including via shared (either multiported or multiplexed) memory, a crossbar switch, a shared bus or an interconnect network of a myriad of topologies including star, ring, tree, hypercube, fat hypercube (a

hypercube with more than one processor at a node), or n-dimensional mesh. Parallel computers based on interconnect networks need to have some kind of routing to enable the passing of messages between nodes that are not directly connected. The medium used for communication between the processors is likely to be hierarchical in large multiprocessor machines.

Cache memory

Cache is a component that transparently stores data so that future requests for that data can be served faster. The data that is stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere. If requested data is contained in the cache (cache hit), this request can be served by simply reading the cache, which is comparatively faster. Otherwise (cache miss), the data has to be recomputed or fetched from its original storage location, which is comparatively slower. Hence, the greater the number of requests that can be served from the cache, the faster the overall system performance becomes.

To be cost efficient and to enable an efficient use of data, caches are relatively small. Nevertheless, caches have proven themselves in many areas of computing because access patterns in typical computer applications have locality of reference. References exhibit temporal locality if data is requested again that has been recently requested already. References exhibit spatial locality if data is requested that is physically stored close to data that has been requested already.

Data is transferred between memory and cache in blocks of fixed size, called cache lines. When a cache line is copied from memory into the cache, a cache entry is created. The cache entry will include the copied data as well as the requested memory location (now called a tag).

When the processor needs to read from or write to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. The cache checks for the contents of the requested memory location in any cache lines that might contain that address. If the processor finds that the memory location is in the cache, a cache hit has occurred. However, if the processor does not find the memory location in the cache, a cache miss has occurred. In the case of:

- a cache hit, the processor immediately reads or writes the data in the cache line
- a cache miss, the cache allocates a new entry, and copies in data from main memory; then, the request is fulfilled from the contents of the cache.

The proportion of accesses that result in a cache hit is known as the hit rate, and can be a measure of the effectiveness of the cache for a given program or algorithm.

The time taken to fetch one cache line from memory (read latency) matters because the CPU will run out of things to do while waiting for the cache line. When a CPU reaches this state, it is called a stall.

As CPUs become faster, stalls due to cache misses displace more potential computation; modern CPUs can execute hundreds of instructions in the time taken to fetch a single cache line from main memory. Various techniques have been employed to keep the CPU busy during this time.

Out-of-order CPUs attempt to execute independent instructions after the instruction that is waiting for the cache miss data.

Another technology, used by many processors, is simultaneous multithreading (SMT), which allows an alternate thread to use the CPU core while a first thread waits for data to come from main memory.

Cache coherency

Cache coherence refers to the consistency of data stored in local caches of a shared resource.

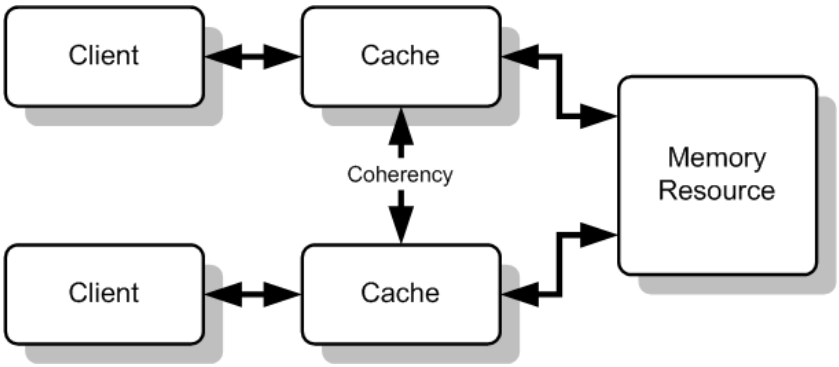


Figure 6 - Multiple Caches of Shared Resource

In a shared memory multiprocessor system with a separate cache memory for each processor, it is possible to have many copies of any one instruction operand: one copy in the main memory and one in each cache memory. When one copy of an operand is changed, the other copies of the operand must be changed also. Cache coherence is the discipline that ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion.

There are three distinct levels of cache coherence:

1. every write operation appears to occur instantaneously
2. all processors see exactly the same sequence of changes of values for each separate operand
3. different processors may see an operation and assume different sequences of values; this is considered to be a non-coherent behavior.

Coherence defines the behavior of reads and writes to the same memory location. The coherence of caches is obtained if the following conditions are met:

1. In a read made by a processor P to a location X that follows a write by the same processor P to X, with no writes of X by another processor occurring between the write and the read instructions made by P, X must always return the value written by P. This condition is related with the program order preservation, and this must be achieved even in monoprocessed architectures.
2. A read made by a processor P1 to location X that follows a write by another processor P2 to X must return the written value made by P2 if no other writes to X made by any processor occur between the two accesses and the read and write are sufficiently separated. This condition defines the concept of coherent view of memory. If processors can read the same old value after the write made by P2, we can say that the memory is incoherent.
3. Writes to the same location must be sequenced. In other words, if location X received two different values A and B, in this order, from any two processors, the processors can never read location X as B and then read it as A. The location X must be seen with values A and B in that order.

These conditions are defined supposing that the read and write operations are made instantaneously. However, this doesn't happen in computer hardware given memory latency and other aspects of the architecture. A write by processor P1 may not be seen by a read from processor P2 if the read is made within a very small time after the write has been made. The memory consistency model defines when a written value must be seen by a following read instruction made by the other processors.[if5]

One of the most common type of coherence mechanism is Directory-based. In a directory-based system, the data being shared is placed in a common directory that maintains the coherence between caches. The directory acts as a filter through which the processor must ask permission to load an entry from the primary memory to its cache. Another known mechanism is Snooping. Snooping is a process where the individual caches monitor address lines for accesses to memory locations that they have cached. It is called a write invalidate protocol when a write operation is observed to a location that a cache has a copy of and the cache controller invalidates its own copy of the snooped memory location.

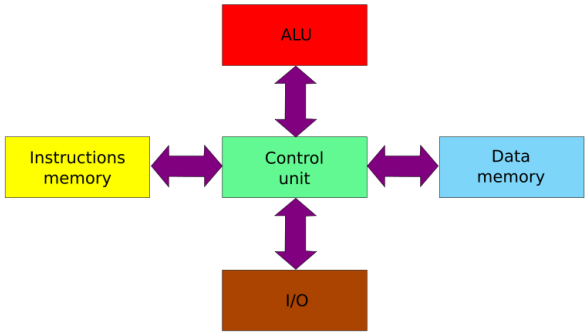
Computer architecture

Harvard architecture

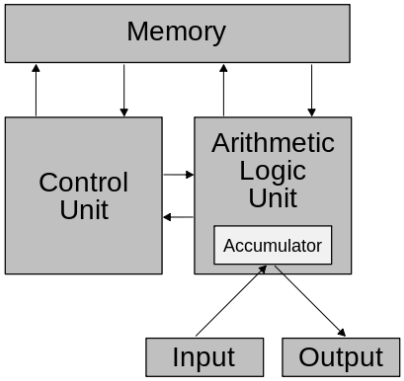
The Harvard architecture is a computer architecture with physically separate storage and signal pathways for instructions and data. Today, most processors implement such separate signal pathways for performance reasons but actually implement a modified Harvard architecture, so they can support tasks such as loading a program from disk storage as data and then executing it.

Von Neumann architecture

A computer with a Von Neumann architecture has the advantage over pure Harvard machines in that code can also be accessed and treated the same as data, and vice versa. This allows, for example, data to be read from disk storage and executed as code, or self-optimizing software systems using technologies such as just-in-time compilation to write machine code into their own memory and then later execute it. Another example is self-modifying code, which allows a program to modify itself. A disadvantage of these methods are issues with executable space protection, which increase the risks from malware and software defects. In addition, in these systems it is notoriously difficult to document code flow, and also can make debugging much more difficult.



Harvard architecture scheme



Von Neumann scheme

Figure 7 -Harvard vs. von Neumann architecture

Modified Harvard architecture

Accordingly, some pure Harvard machines are specialty products. Most modern computers instead implement a modified Harvard architecture. Those modifications are various ways to loosen the strict separation between code and data, while still supporting the higher performance concurrent data and instruction access of the Harvard architecture.

Split cache architecture

The most common modification builds a memory hierarchy with a CPU cache separating instructions and data. This unifies all except small portions of the data and instruction address spaces, providing the von Neumann model. Most programmers never need to be aware of the fact that the processor core implements a (modified) Harvard architecture, although they benefit from its speed advantages. Only programmers who write instructions into data memory need to be aware of issues such as cache coherency and executable space protection.

Access instruction memory as data

Another change preserves the "separate address space" nature of a Harvard machine, but provides special machine operations to access the contents of the instruction memory as data. Because data is not directly executable as instructions, such machines are not always viewed as "modified" Harvard architecture:

Read access: initial data values can be copied from the instruction memory into the data memory when the program starts. Or, if the data is not to be modified (it might be a constant value, such as pi, or a text string), it can be accessed by the running program directly from instruction memory without taking up space in data memory (which is often at a premium).

Write access: a capability for reprogramming is generally required; few computers are purely ROM based. For example, a microcontroller usually has operations to write to the flash memory used to hold its instructions. This capability may be used for purposes including software updates and EEPROM replacement.

Instruction set

An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor.

Classification of instruction sets

A complex instruction set computer (CISC) has many specialized instructions, which may only be rarely used in practical programs. A reduced instruction set computer (RISC) simplifies the processor by only implementing instructions that are frequently used in programs; unusual operations are implemented as subroutines, where the extra processor execution time is offset by their rare use.

Very long instruction word (VLIW)

Very long instruction word or VLIW refers to a processor architecture designed to take advantage of instruction level parallelism (ILP). Whereas conventional processors mostly only allow programs that

specify instructions to be executed one after another, a VLIW processor allows programs that can explicitly specify instructions to be executed at the same time (i.e. in parallel). This type of processor architecture is intended to allow higher performance without the inherent complexity of some other approaches. One VLIW instruction encodes multiple operations; specifically, one instruction encodes at least one operation for each execution unit of the device. For example, if a VLIW device has five execution units, then a VLIW instruction for that device would have five operation fields, each field specifying what operation should be done on that corresponding execution unit. To accommodate these operation fields, VLIW instructions are usually at least 64 bits wide, and on some architectures are much wider.

VLIW CPUs use *software* (the compiler) to decide which operations can run in parallel. Because the complexity of instruction scheduling is pushed off onto the compiler, the hardware's complexity can be substantially reduced.

Performance improving

Traditional approaches to improving performance in processor architectures include breaking up instructions into sub-steps so that instructions can be executed partially at the same time (pipelining), dispatching individual instructions to be executed completely independently in different parts of the processor (superscalar architectures), and even executing instructions in an order different from the program (out-of-order execution). These approaches all involve increased hardware complexity (higher cost, larger circuits, higher power consumption) because the processor must intrinsically make all of the decisions internally for these approaches to work.

Instruction pipeline

An instruction pipeline is a technique used in the design of computers to increase their instruction throughput (the number of instructions that can be executed in a unit of time). Pipelining does not reduce the time to complete an instruction, but increases instruction throughput by performing multiple operations in parallel.

Each instruction is split into a sequence of dependent steps. The first step is always to fetch the instruction from memory; the final step is usually writing the results of the instruction to processor registers or to memory. Pipelining lets the computer's cycle time be the time of the slowest step, and ideally lets one instruction complete in every cycle.

The term pipeline is an analogy to the fact that there is fluid in each link of a pipeline, as each part of the processor is occupied with work.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figure 8 - Basic five-stage pipeline

Figure 10 illustrates the basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

Central processing units (CPUs) are driven by a clock. Each clock pulse need not do the same thing; rather, logic in the CPU directs successive pulses to different places to perform a useful sequence. There are many reasons that the entire execution of a machine instruction cannot happen at once. For example, if one clock pulse latches a value into a register or begins a calculation, it will take some time for the value to be stable at the outputs of the register or for the calculation to complete. As another example, reading an instruction out of a memory unit cannot be done at the same time that an instruction writes a result to the same memory unit. In pipelining, effects that cannot happen at the same time are made the dependent steps of the instruction.

Hazards

A human programmer writing an assembly language program on the sequential-execution model assumes that each instruction completes before the next one begins. This assumption is not true on a pipelined processor. A situation where the expected result is problematic is a hazard. Imagine the following two register instructions to a hypothetical RISC processor:

```
1: Add 1 to R5.
2: Copy R5 to R6.
```

If the processor has the 5 steps listed in the initial illustration, instruction 1 would be fetched at time t1 and its execution would be complete at t5. Instruction 2 would be fetched at t2 and would be complete at t6. The first instruction might deposit the incremented number into R5 as its fifth step (register write back) at t5. But the second instruction might get the number from R5 (to copy to R6) in its second step (instruction decode and register fetch) at time t3. It seems that the first instruction would not have incremented the value by then. The above code invokes a hazard.

A human programmer writing in a compiled language might not have these concerns, as the compiler could be designed to generate machine code that avoids hazards.

Work-arounds

In order to avoid hazards, the programmer may have unrelated work that the processor can do in the meantime; or, to ensure correct results, the programmer may insert NOPs into the code, partly negating the advantages of pipelining.

Solutions

Pipelined processors commonly use three techniques to work as expected when the programmer assumes that each instruction completes before the next one begins:

- Processors that can compute the presence of a hazard may stall — delay processing of the second instruction (and subsequent instructions) until the values it requires as input are ready. This creates a bubble in the pipeline, also partly negating the advantages of pipelining.
- Some processors can not only compute the presence of a hazard but can compensate by having additional data paths that provide needed inputs to a computation step before a subsequent instruction would otherwise compute them, an attribute called forwarding.
- Some processors can determine that instructions other than the next sequential one are not dependent on the current ones and can be executed without hazards. Such processors may perform out-of-order execution, which will be discussed in the next subsection.

Branches

A branch out of the normal instruction sequence often involves a hazard. Unless the processor can give effect to the branch in a single time cycle, the pipeline will continue fetching instructions sequentially. Such instructions cannot be allowed to take effect because the programmer has diverted control to another part of the program.

A conditional branch is even more problematic. The processor may or may not branch, depending on a calculation that has not yet occurred. Various processors may stall, may attempt branch prediction, and may be able to begin to execute two different program sequences (eager execution), both assuming the branch is and is not taken, discarding all work that pertains to the incorrect guess.

There are two subtopics related to this section: branch prediction and branch target prediction; branch prediction attempts to guess whether a conditional jump will be taken or not, while the latter attempts to guess the target of a taken conditional or unconditional jump before it is computed by decoding and executing the instruction itself. Branch prediction and branch target prediction are often combined into the same circuitry.

A processor with an implementation of branch prediction that usually makes correct predictions can minimize the performance penalty from branching. However, if branches are predicted poorly, it may create more work for the processor, such as flushing from the pipeline the incorrect code path that has begun execution before resuming execution at the correct location. Hence, programs written for a pipelined processor deliberately avoid branching to minimize possible loss of speed. For example, the programmer can handle the usual case with sequential execution and branch only on detecting unusual cases.

An illustrated example

Figure 11 depicts a generic pipeline with four stages as described in the picture.

The top gray box is the list of instructions waiting to be executed; the bottom gray box is the list of instructions that have been completed; and the middle white box is the pipeline. The colored boxes represent instructions independent of each other.

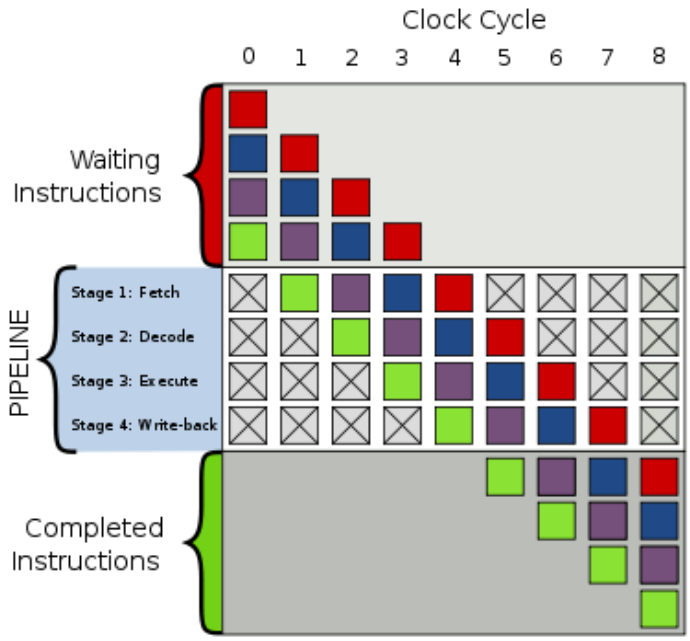


Figure 9 - Generic 4-stage pipeline

Execution is as follows:

Time	Execution
0	Four instructions are waiting to be executed
1	• The green instruction is fetched from memory
2	• The green instruction is decoded • The purple instruction is fetched from memory
3	• The green instruction is executed (actual operation is performed) • The purple instruction is decoded • The blue instruction is fetched
4	• The green instruction's results are written back to the register file or memory • The purple instruction is executed • The blue instruction is decoded • The red instruction is fetched
5	• The green instruction is completed • The purple instruction is written back • The blue instruction is executed • The red instruction is decoded
6	• The purple instruction is completed • The blue instruction is written back • The red instruction is executed
7	• The blue instruction is completed • The red instruction is written back
8	• The red instruction is completed
9	All four instructions are executed

A bubble in the pipeline

A pipelined processor may deal with hazards by stalling and creating a bubble in the pipeline, resulting in one or more cycles in which nothing useful happens.

In figure 11, in cycle 3, the processor cannot decode the purple instruction, perhaps because the processor determines that decoding depends on results produced by the execution of the green instruction. The green instruction can proceed to the Execute stage and then to the Write-back stage as scheduled, but the purple instruction is stalled for one cycle at the Fetch stage. The blue instruction, which was due to be fetched during cycle 3, is stalled for one cycle, as is the red instruction after it.

Because of the bubble (the blue ovals in the illustration), the processor's Decode circuitry is idle during cycle 3. Its Execute circuitry is idle during cycle 4 and its Write-back circuitry is idle during cycle 5. When the bubble moves out of the pipeline (at cycle 6), normal execution resumes. But everything now is one cycle late. It will take 8 cycles (cycle 1 through 8) rather than 7 to completely execute the four instructions shown in colors.

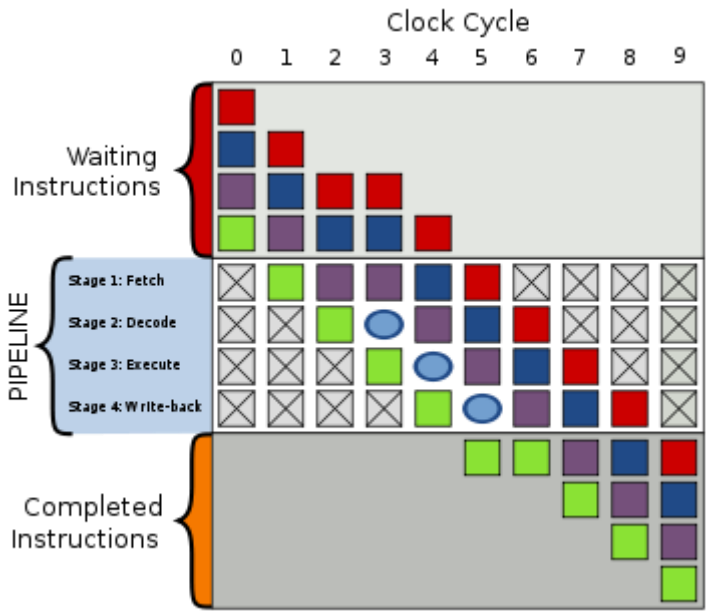


Figure 10 - A bubble in cycle 3 delays execution

Superscalar

Superscalar describes a microprocessor design that makes it possible for more than one instruction at a time to be executed during a single clock cycle. In a superscalar design, the processor or the instruction compiler is able to determine whether an instruction can be carried out independently of other sequential instructions, or whether it has a dependency on another instruction and must be executed in sequence with it. The processor then uses multiple execution units to simultaneously carry out two or more independent instructions at a time.

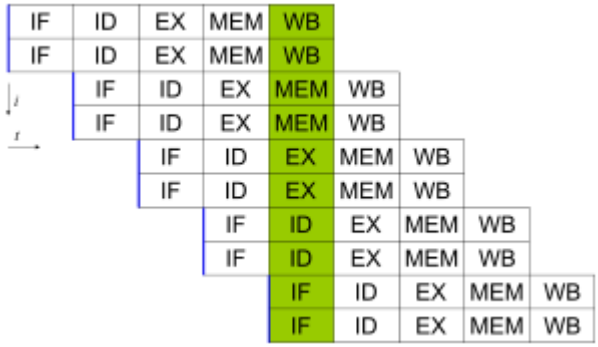


Figure 11 - Simple superscalar pipeline

In a superscalar CPU the dispatcher reads instructions from memory and decides which ones can be run in parallel, dispatching them to redundant functional units contained inside a single CPU. Therefore a superscalar processor can be envisioned having multiple parallel pipelines, each of which is processing instructions simultaneously from a single instruction thread. Figure 13 illustrates a simple superscalar pipeline. By fetching and dispatching two instructions at a time, a maximum of two instructions per cycle can be completed.

Superscalar processors differ from multi-core processors in that the redundant functional units are not entire processors. A single processor is composed of finer-grained functional units such as the ALU, integer multiplier, integer shifter, floating point unit, etc. There may be multiple versions of each functional unit to enable execution of many instructions in parallel. This differs from a multi-core processor that concurrently processes instructions from multiple threads, one thread per core. It also differs from a pipelined CPU, where the multiple instructions can concurrently be in various stages of execution, assembly-line fashion.

The various alternative techniques are not mutually exclusive—they can be combined in a single processor. Thus a multicore CPU is possible where each core is an independent processor containing multiple parallel pipelines, each pipeline being superscalar. Some processors also include vector capability.

Simultaneous multithreading (SMT)

Abbreviated as SMT, simultaneous multithreading is a processor design technology that allows multiple threads to issue instructions each cycle. Simultaneous multithreading enables multithreaded applications to execute threads in parallel on a single multi-core processor instead of processing threads in a linear fashion.

Out-of-order execution (OoOE or OOE)

Out-of-order execution (OoOE) is an approach to processing that allows instructions for high-performance microprocessors to begin execution as soon as their operands are ready. Although instructions are issued in-order, they can proceed out-of- order with respect to each other.

Out-of-order processors break up the processing of instructions into these steps:

1. Instruction fetch.

2. Instruction dispatch to an instruction queue (also called instruction buffer or reservation stations).
3. The instruction waits in the queue until its input operands are available. The instruction is then allowed to leave the queue before earlier, older instructions.
4. The instruction is issued to the appropriate functional unit and executed by that unit.
5. The results are queued.
6. Only after all older instructions have their results written back to the register file, then this result is written back to the register file. This is called the graduation or retire stage.

The goal of OoO processing is to allow the processor to avoid a class of stalls that occur when the data needed to perform an operation are unavailable.

Speculative execution

The ability to issue instructions past branches which have yet to resolve is known as speculative execution. It is an optimization technique where a computer system performs some task that may not be actually needed. The main idea is to do work before it is known whether that work will be needed at all, so as to prevent a delay that would have to be incurred by doing the work after it is known whether it is needed. If it turns out the work was not needed after all, any changes made by the work are reverted and the results are ignored.

Dataflow architecture

Dataflow architecture is a computer architecture that directly contrasts the traditional von Neumann architecture or control flow architecture. Dataflow architectures do not have a program counter, or the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable: i. e. behavior is indeterministic. This architecture wasn't commercially successful mainly because of the challenging debug process of programs running on such architecture. Despite that dataflow architectures that are deterministic in nature enable programmers to manage complex tasks such as processor load balancing, synchronization and accesses to common resources.

Part C – The HyperCore Architecture[if6]

Processor Architecture

The HyperCore architecture is designed to handle a wide range of general-purpose applications speedily and efficiently. Plurality's HyperCore Architecture Line (HAL) family of massively parallel processors implement the HyperCore architecture. The HAL family includes 16 to 256 32-bit RISC cores, a shared memory architecture, and a hardware-based scheduler that supports a task-oriented programming model. HAL is designed to offer unequaled performance capabilities over a wide range of applications while providing “near serial” programmability, hiding from the programmer the complexity of the massively parallel processing operation.

HAL cores are compact 32-bit RISC cores, which execute a subset of the SPARC v8 instruction set with extensions. These cores have been designed specifically to work in a massively parallel environment. A unique feature of the HAL architecture is the shared memory system, which allows an instruction fetch and a data access by each individual core at each clock cycle. The processors do not have any private cache or memory, avoiding coherency problems.

The role of the scheduler is to allocate tasks to cores. It employs a configurable matrix that defines task dependencies, and it monitors the state of each core. Combining these two inputs, it allocates tasks while also maintaining a dynamic load balance in runtime. Allocated tasks are executed to completion before their cores become available for new allocations.

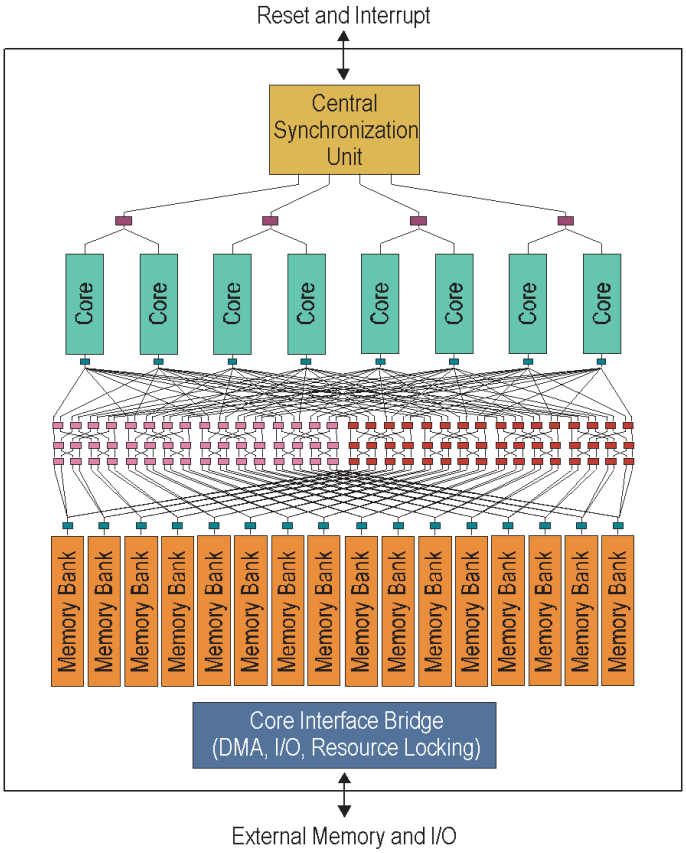


Figure 12 - The HyperCore Architecture

Figure 14 shows the basic architecture of an eight-core implementation. It has a hardware synchronization and scheduling unit, eight 32-bit RISC cores, shared on-chip memory that is accessed through a high-performance interconnect network, and a core interface bridge that mediates between the internal and external environments. Note that only the data memory is shown; it does not illustrate the instruction memory or a cache implementation of shared memory.

The architecture supports a task-oriented parallel programming model that enhances the traditional serial programming model with simple constructs for handling massively parallel operations. The extremely low overhead for enabling tasks and dispatching them to cores allows finer-grained parallelism than is possible when task scheduling and synchronization are performed by operating-system software. A guiding principle of the architecture was to avoid features that give poor return per unit of silicon area and power budget, such as private (per-core) memory, cache coherency, interprocessor communication, speculative execution, superscalar pipelines, and private (per-core) routers.

HyperCore Implementations

Implementations of the HyperCore architecture have these parts:

- 16 to 256 32-bit RISC processor cores.
- 4 to 64 coprocessors that include a floating-point unit and a divider. Each coprocessor is shared by a configurable number of cores, typically 4 cores per coprocessor.
- Shared memory system (operating as a cache or as simple random-access memory).
- Hardware synchronizer/scheduler.
- Core interface bridge to communicate with I/O and internal shared resources.

The entire shared-memory linear address space is available to all cores. The shared memory holds program, data, stack, and dynamically allocated memory. Each core has two memory ports: an instruction port that can read memory, and a data port that can either read or write memory. Both ports can operate simultaneously. The shared memory system has a hardware conflict-resolution mechanism. During each clock cycle, most cores can perform one instruction read and one data read/write access. Conflicting accesses are relatively infrequent and are resolved on-the-fly; unserviced memory-access requests are delayed for a cycle. An access can experience more than one cycle of delay, but this happens only rarely and causes only slight degradation of aggregate performance. Plurality's cycle-accurate simulator predicts the exact performance of any specific application.

The hardware synchronizer/scheduler consists of two parts: the Central Synchronization Unit (CSU) and the Distribution Network (DN), as shown in Figure 12. The CSU is responsible for allocating tasks to cores based on a preloaded task map designed by the programmer. It also collects completion indications and monitors the activity status of each core. The CSU optimizes throughput and latency of task allocation and eliminates bottlenecks, which is critical for parallel processing performance. Tasks that can run as multiple instances, in parallel, are allocated by the CSU in packs (sets) of task instances.

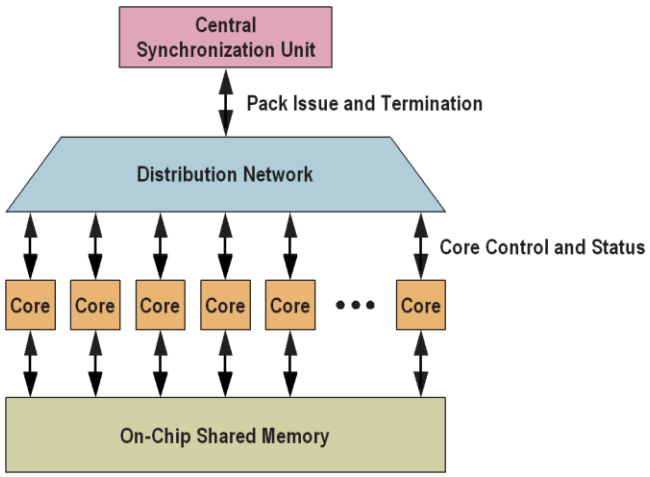


Figure 13 - HyperCore Parallel Architecture

The DN interfaces the CSU to the cores. It distributes allocated packs of task instances from the CSU to the cores, unifies task-termination events from the cores to the CSU, and updates the number of cores available for allocation by the CSU.

Cores

Implementations of the HyperCore architecture have a configurable number of processing cores—tens or hundreds of cores. Figure 16 shows a single core. All cores are identical, and each can run any task. The numbers of cores can be scaled without affecting application software.

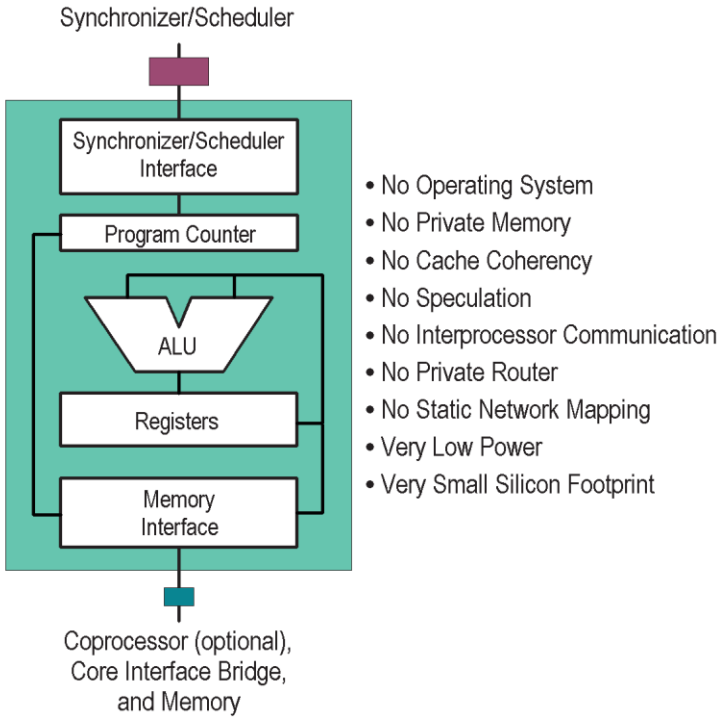


Figure 14 - Core Architecture

Each core includes:

- Register file
- Arithmetic logic unit
- Instruction and data pipeline
- Memory interface
- Coprocessor interface
- Debug interface

The cores do not run an operating system, they do not use interprocessor communication, and they do not have private memory, a private network router, cache-coherency logic, or instruction-speculation logic. They are very compact in size and consume very little power.

The hardware synchronizer/scheduler and shared memory architecture eliminate any requirement for explicit management of the communication between the cores. When a core is allocated for executing a task, it runs serially until completion. A core does not have to know what the other cores are executing nor the current state of the entire machine.

Coprocessors

The coprocessor approach efficiently shares larger functional blocks (floating-point and divider units) among the cores. Each coprocessor is shared by a configurable number of cores, typically 4 cores, as shown in Figure 17.

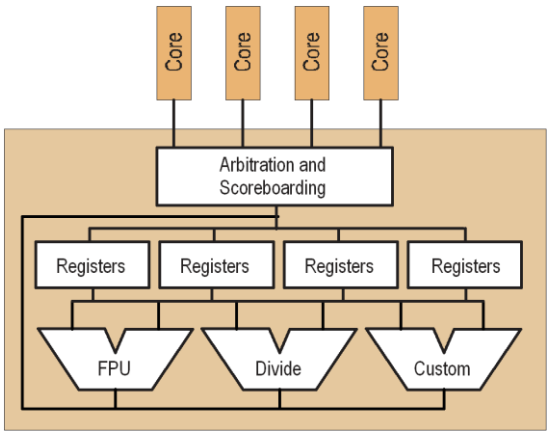


Figure 15 - Coprocessor Architecture

Each core has a private register file in its coprocessor dedicated for use by that core. Each coprocessor has its own pipeline and locking mechanism to prevent pipeline hazards. This allows each core to operate the coprocessor in parallel with its own pipeline. Register contents can be transferred between the core's main register file and its coprocessor register file when required. The compiler manages use of the coprocessor transparently to the application software. Different coprocessors are available and are customizable.

The basic implementation includes:

- Divider—32-bit x 32-bit to a 32-bit quotient, non-pipelined, variable latency.
- Floating-Point Unit—Multiplication, addition/subtraction and conversions with pipelined 4-cycle latency. Division and square root are non-pipelined with variable latency.[if7]

Central Synchronization Unit

The hardware Synchronizer/Scheduler dynamically manages task allocation, task scheduling, and synchronization among the parallel processing cores. It uses a task-dependency graph and monitors the state of each core. With this information, it allocates tasks to cores at hardware speed in a way that dynamically maintains load balance among the cores. It takes the place of an operating system's scheduling and synchronization functions. There is no operating system running in the cores or elsewhere on the chip, so this software overhead is eliminated. Instead of hundreds or thousands of cycles per task allocation, only a few cycles are needed. The Synchronizer/Scheduler consists of two parts: a Central Synchronization Unit and a Distribution Network. Figure 5 shows a 32-core implementation of the Synchronizer/ Scheduler.

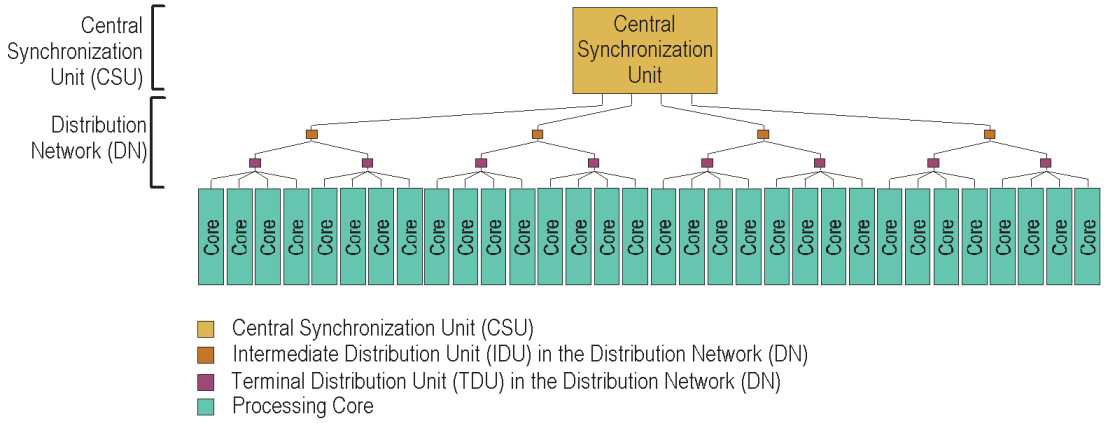


Figure 16 - Synchronizer / Scheduler

The Central Synchronization Unit manages allocation, scheduling, and synchronization of tasks. It also collects task-completion information and monitors the activity status of each core. It contains a hardware matrix that is programmed by software to represent the task-dependency graph of a parallel application program. It then allocates packs (sets) of parallel task-instances to the Distribution Network, which distributes the instances to individual cores. Being implemented in hardware, it avoids time-consuming software synchronizing primitives, such as test-and-set and fetch-and-add, that do not scale well and that prevent fine-grained concurrency. Throughput and latency of task allocation are very fast, eliminating bottlenecks that impede other massively parallel architectures.

The Distribution Network is a multistage logarithmic network in a forest topology (a set of trees) with the Central Synchronization Unit as its root. It mediates between the Central Synchronization Unit and the processing cores, and vice versa. In its downstream flow, the Distribution Network decomposes allocated packs of task instances and delivers them to the available cores. In its upstream flow, it unifies task-

termination events from the cores and updates the number of cores available for new tasks. The Synchronizer/Scheduler can maintain a throughput of four task-pack allocations per cycle and four task-pack terminations per cycle. The time to allocate a new task to a core is very short. In a 64-core configuration, for example, it is typically 5 to 9 clock cycles (Figure 19), depending on which of four groups the core is in. This is the time from enabling the task to the time that a particular core, running a particular task instance, fetches the task's first instruction. Termination of the last instance of a task takes an additional five cycles.

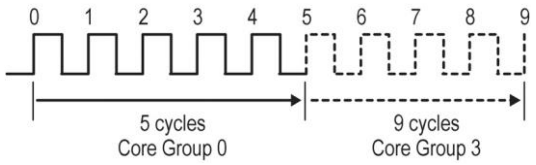


Figure 17 - Task allocation time

The Synchronizer/Scheduler can be configured to support any number of tasks and can manage hundreds of tasks and cores in parallel. There is no static mapping of tasks to cores. Any available core can execute an instance of any enabled task. When the number of parallel tasks exceeds the number of cores, the Synchronizer/Scheduler automatically schedules the tasks according to pre-determined priority and the number of available cores. This scheduling method ensures that code written for one implementation of the HyperCore architecture transparently scales in performance when run on an implementation with a larger or smaller number of cores.

Shared Memory

As we seen earlier - the performance of most digital systems today is limited by the performance of their interconnection between logic and memory, rather than the performance of the logic or memory itself. The HyperCore architecture solves this problem by using a high-performance network to interconnect its processing cores and on-chip shared memory banks.

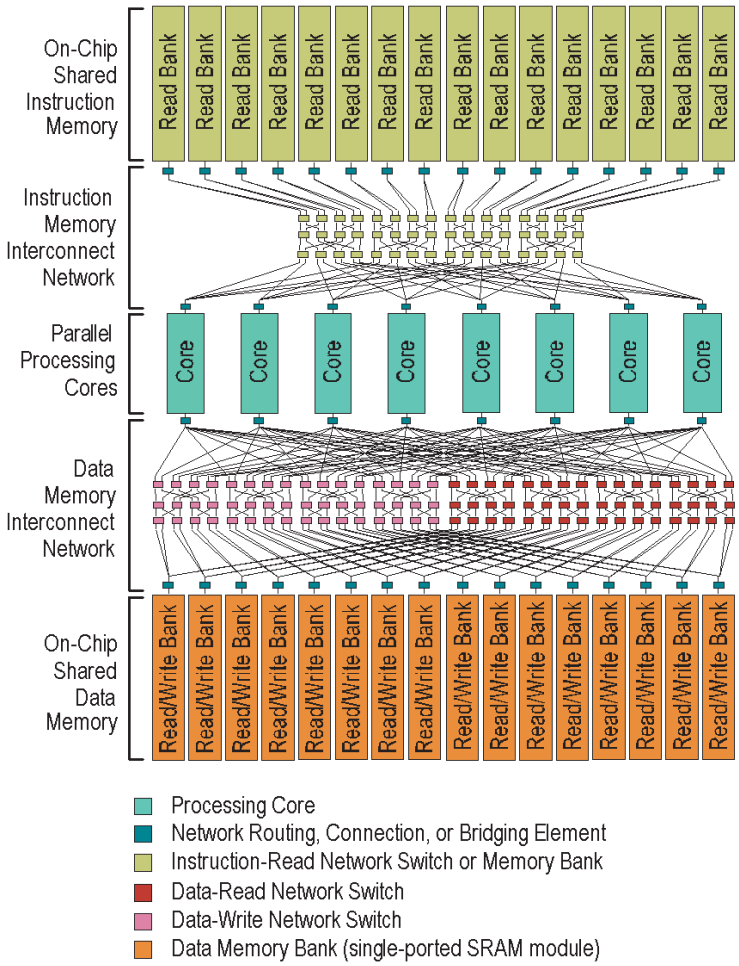


Figure 18 - Shared Memory and Interconnect Network

The interconnect network is structured much like those found in heavily trafficked data centers. It is a multistage logarithmic network in which every path from a core to a memory bank passes through a series of uniform switches. It is essentially a combinational circuit in which data, addresses, and control signals move at hardware speed through the switches.

There is no static mapping of the network, and the network has no buffering, except for implementations that use registers to implement a pipeline. The network shown in Figure 17 for an eight-core implementation consists of two separate parts, one for reading instructions (shown above the cores) and one for reading or writing data (shown below the cores). Each core has two memory ports, one for instructions and one for data. Both ports, and their networks, operate simultaneously. During each clock cycle, most cores can perform one instruction read and one data read or write.

Each core has two ports to a shared memory, a read-only instruction port and a read/write data port. The shared memory allows all of the cores to access memory efficiently in parallel without being concerned about memory coherency. Data written from one core is available to all cores when the write completes, however a proper program does not allow cores to update each other's currently used memory. Synchronization is performed at the task level, not at the level of shared memory, so the results produced by one core must not be read by any other core until after the original core has terminated. All cores can read the same memory location simultaneously without suffering from a bottleneck.

The implementation of shared memory allows the HyperCore architecture to obtain excellent results when executing fine-grained parallel programs, because all data is accessible to all cores without any synchronization or coherency overhead. From the developer's point-of-view, the memory model is like that of a unicore processor.

Since the entire address space of shared memory is available to all cores, and all cores are equidistant from memory - there is no per-core private memory, so there is no memory-coherence overhead. All cores can read memory in the same clock cycle without suffering a bottleneck. Data written by one core can be read by any core at the end of the writing sequence. Only simple reads and writes are supported. A hardware mutual-exclusion facility locks resources, such as the DMA controller's registers, that cannot be used by more than one core at a time. The instruction network and the read part of the data network support multicast reads, as shown in Figure 4. This multicast function enables all, or any subgroup, of the cores to read the same memory address in the same cycle, thereby preventing read hot-spots.

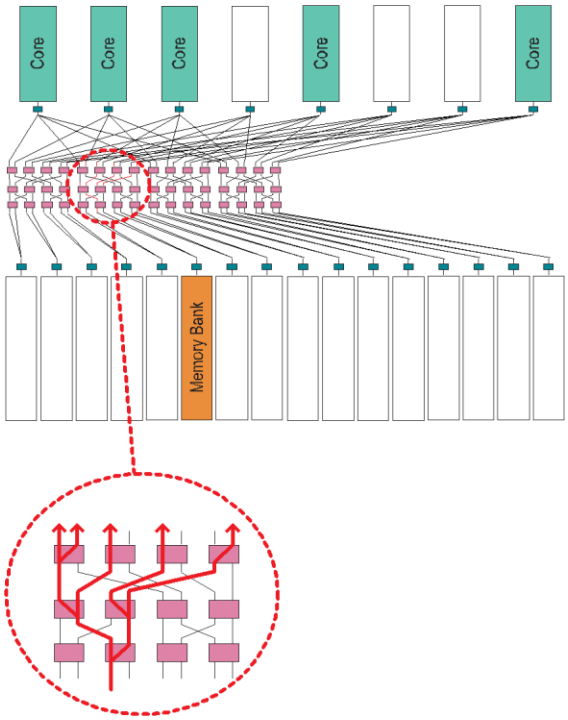


Figure 19 - Multicast read

The paths between cores and memory banks change on each clock cycle to satisfy the changing read and write requests from the cores. All path intersections are switches that change with combinational memory-access signals. Thus, the network is access-switched through each part of its path, not circuit-switched end-to-end. The network computes a path using the memory address and access type (read or write) provided by the core. This computed path defines the memory bank, the address in that memory bank, and the network switching. If two cores access the same address in shared memory, they refer to the same bank and same address inside that bank, but their network paths are different because the endpoints (the cores) are different. Memory addresses are interleaved among memory banks, such that addresses very close to one another are implemented in different memory banks; this is called *anti-local address mapping*, and it reduces memory-access conflicts. A conflict between two requested paths occurs

on data reads or writes when both accesses pass through the same port of the same crossbar switch (or other basic network element) but are directed at different addresses. These conflicts are detected locally at the switch. The switch then makes a local decision to grant access to one of the contesting cores, based on fixed priorities among cores. If a downstream (toward memory) switch makes a different local decision, that decision is forwarded upstream and overrides decisions of upstream switches. The number of network paths and memory banks can be tuned to reduce these conflicts to a negligible level.

Coarse vs. Fine-Grain Parallelism

An important design goal for the HyperCore architecture was moving task scheduling and synchronization out of the OS domain into hardware. This eliminates overhead for task allocation to cores, in stark contrast to using the OS which costs hundreds or thousands of instruction cycles per task allocation. Fast scheduling and synchronization allows exploiting forms of parallelism unavailable to traditional approaches, because per-task overhead sets a lower limit on the size of opportunities that can be profitably handled in parallel. A 100-instruction loop iteration may not be optimizable with parallel processing if it costs more than 100 instruction cycles to allocate a task to a core. In the HyperCore architecture, task allocations cost far less than 100 cycles.

The HyperCore architecture relies on the existence of “duplicable tasks” within programs. These are sections of code which are executed repetitively against different sets of data. They are common in computationally intensive applications like video processing, pattern recognition, graphics rendering, and wireless communications. A single core executes a “task instance” of a duplicable task. Task instances are allocated to cores for parallel execution, but all instances do not necessarily execute at the same time. If the number of task instances exceeds the number of cores, they will be automatically scheduled in groups. Even if the number of instances does not exceed the number of cores, there is no guarantee on the order in which task instances are allocated to cores or the order in which they finish execution. (A hardware mutual-exclusion lock facility is provided for protecting critical resources like access to the control and status registers of a DMA controller and calls to the memory manager software.)

Code granularity is the size of the code executed in a task instance. A fine-grained parallel program running on an inefficient scheduler will run slower than a serial implementation of the same program. A coarse-grained task is usually considered one with hundreds or more machine instructions—a thread in today’s operating systems. A fine-grained task is one with 20to 100 machine instructions. The HyperCore architecture does not limit the size of these code sections; it efficiently handles them all. Fine-grained parallel programs execute efficiently in the HyperCore architecture. This is due to the task-allocation speed of the hardware Synchronizer/Scheduler, the access speed of shared on-chip memory, and the lack of memory-coherency and operating-system overhead. Converting a serial program into a fine-grained parallel program increases the demand for task allocation and synchronization between parallel tasks. The Synchronizer/Scheduler supports fine-grained programs because large numbers of free cores can be allocated task instances in parallel, very quickly. Task allocation typically takes between 5 and 9 clock cycles (Figure 17). By contrast, conventional parallel architectures perform task allocation in software, where much greater overhead makes parallel execution of short code sequences inefficient. Shared memory supports fine-grained programs because all data is accessible to all cores with minimal overhead. An individual core’s memory model is like that of a uniprocessor, in which it has access to all memory.

Task-Oriented Programming

Unlike conventional processor architectures in which a single reset signal initiates execution of a program which directly or indirectly spawns all subsequent tasks, the HyperCore architecture defines many tasks which lie dormant until activated by one of many hardware signals. These signals are implemented in the Synchronizer/Scheduler's hardware matrix, which interprets a prioritized task-scheduling model of the application and monitors the activity of cores. For each task, the hardware matrix decides how to allocate, to the available cores, some or all of the task instances available for execution. The matrix also responds to task-termination signals from cores by enabling subsequent tasks. The HyperCore task-oriented programming model will be familiar to programmers acquainted with threads or parallel algorithms. Many recently developed programs use threads (or "tasks", as Plurality calls them), but their use is typically limited to just a few threads—for example, a compute thread, a rendering thread that displays graphic results, and a user-interface thread. However, multicore processors work most efficiently when there are many tasks in a task-allocation pipeline, and there are many task instances running in parallel on many cores. Most processors lack efficient synchronization resources to support large numbers of parallel threads, because the operating system is slow at allocating and switching threads, data dependencies require threads to be kept on a single processor, and atomic operations, deadlock conditions, or events become too complex to manage in software. The HyperCore programming model provides a simple method for converting serial programs into parallel programs. *There are no thread pools or work queues that cores draw from. There is no message-passing between cores. There is usually no need to logically restructure serial program code, except when restructuring can greatly increase performance such as by configuring tasks in pipelines.*

A programmer begins by designing a dependency graph for source-code blocks (such as inner loops) that can run in parallel as independent task instances. Then, the names of some variables, statements, and functions are changed to identify parallel operations and implement the dependency graph. Finally, the revised source code is compiled to generate (1) a task map, which is stored in and executed by the Synchronizer/Scheduler, and (2) task binaries, which are stored in shared memory and executed by the cores.

A task map is a description of the control-flow dependency graph that is used by the Synchronizer/Scheduler to schedule tasks. Figure 20 shows an example. The topology of the graph shows which code blocks must precede other blocks. The blocks consist of both serial tasks, which must run on only one core, and parallel tasks, which can run on any number of available cores simultaneously. The transitions between blocks become task-allocation events that are managed by the Synchronizer/Scheduler. To port an existing sequential program to the HyperCore architecture, software developers convert inner loops into duplicable tasks (by substituting a few names and adding a little bit of syntax), and they write a task map or code that makes a task map. A task map is derived from a dependency graph, like that shown in Figure 22, which shows the order in which tasks execute.

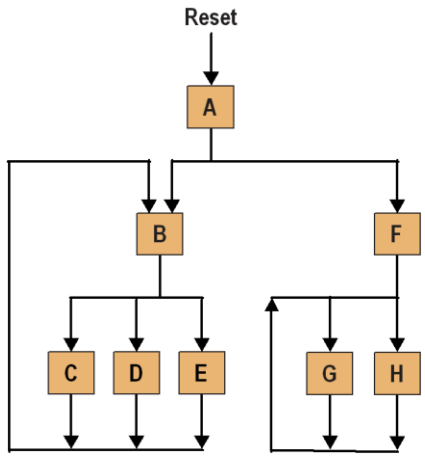


Figure 20 - Dependency Graph Used To Develop a Task Map

To start thinking in task-oriented programming terms, a developer must realize that the program no longer has a single entry point, such as main() or any equivalents. The program's flow is no longer determined in the body of that entry point, and the program is no longer terminated when the entry point returns control to the operating system. The program's flow is defined in the higher level of a predefined, or dynamically created, task map.

The task map provides a clean, simple, easy-to-change method for representing parallel program flow. It controls which task or tasks should be executed after system initialization and controls the relationship between the termination of one task and the spawning of another. If, for example, the developer wants to start task B after the termination of task A, he or she simply indicates that dependency relationship in the task map, where other developers can see and read the program's flow.

Task Types

Broadly speaking, a task is an executing program. In a conventional uniprocessor programming paradigm, the OS sets up a task and grants time-slots for its execution. In the Task Oriented Programming paradigm (TOP), a hardware signal enables execution of the task, which is then allocated by the hardware synchronizer/scheduler to a core, runs to completion, and terminates. When the task terminates, it sends tokens, which are hardware signals that can enable the execution of other tasks.

In the TOP paradigm, we will discuss these types of tasks:

- *Regular task*—runs a single thread sequentially on a single core. For more complex program flow, regular tasks can affect control flow by three types of termination conditions asserted by user's code (true, false, and don't care).
- *Duplicable task*—can be executed in parallel as multiple instances on multiple cores. Amount of instances (Quota) is programmable during a previously allocated task. If Quota is zero, the task terminates immediately after enabling. Each core receive an instance number to identify which instance should be executed. All instances of the task must terminate before a termination token is generated. A duplicable task only indicates termination. A duplicable task can be assigned a static priority.
- *Dummy task*—not allocated to a core and terminates immediately when enabled. Dummy tasks are used to allow more complex task enabling conditions. Like duplicable tasks, dummy tasks only indicate termination.

Implementing a Task-Oriented Solution

Consider an application like that shown in Figure 23. Following reset, an initialization section sets up all of the data structures needed by the application and launches an engine that reads input data, processes it, and passes it to an output. In the figure, this is shown as three sequential processes A, B, and C.

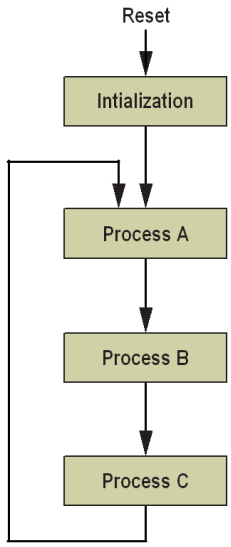


Figure 21 - Application breakdown into processes

A problem of this type, implemented in the Task Oriented Programming, may look like Figure 22. At reset, a special task called “initialization task 0” executes. When that task terminates, it enables the first task in Process A.

A process that can be made parallel is executed as a duplicable task. The duplicable task terminates when its last instance terminates. In Figure 22, the last task instance of Process A enables all of the task instances of Process B, and the last task instance of Process B enables Process C. Task instances may terminate in any order, so instance N-1 of N instances is not necessarily the first or last task instance to terminate. Software must not make any assumptions about the order in which task instances are executed.

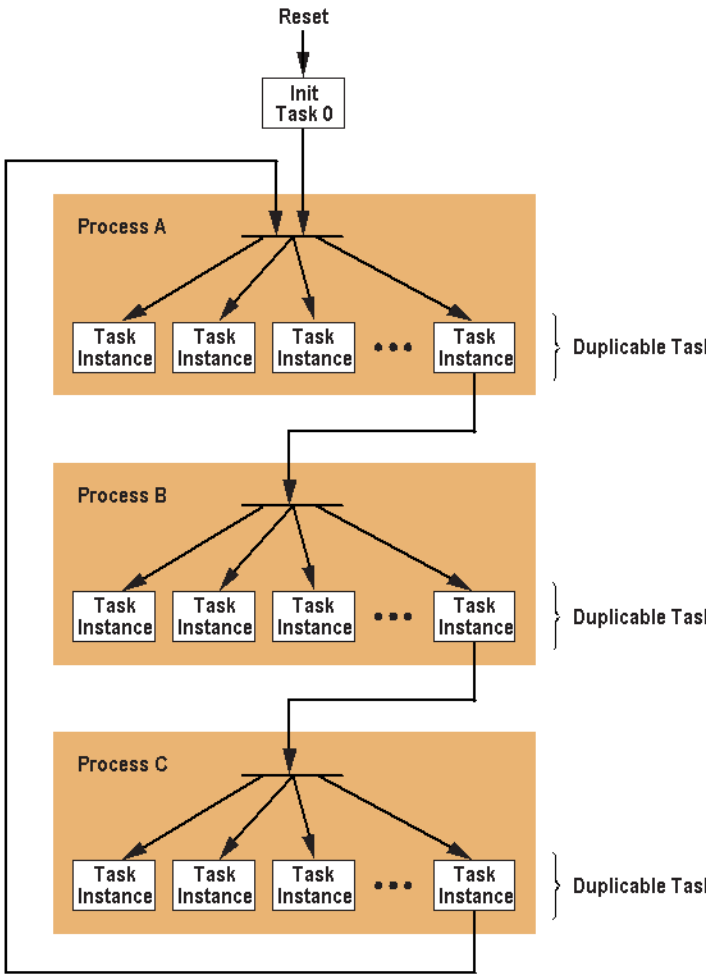


Figure 22 - Duplicable tasks execute as parallel task instances

Tokens

Termination of a task causes tokens to be sent to each of its dependent tasks. To enable a task, its input tokens must be received so its enabling condition becomes true. For example, if task B is dependent on termination of task A, task A passes a token to task B when it terminates, and task B receives a token from task A. The conditions are usually simple, but can be complex.

Hardware supports five types of task-enabling conditions, as shown in the table below:

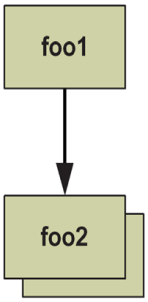
Condition	Description
A	Enabled when task A terminates.
A | B	Enabled when task A or B terminates. This is an OR condition
A & B	Enabled when both task A and B terminate. This is and AND condition.
(A | B) & C	Enabled when either (task A or task B) and task C terminate.
Empty	Enabled after the initialization of the program. This is called pre-enabled condition.

Graphing Task Dependencies

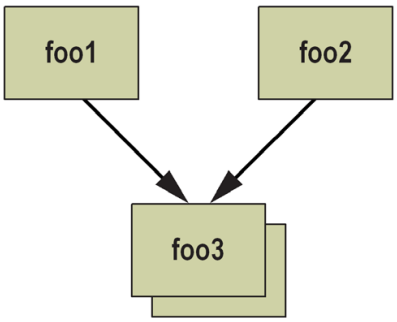
A dependency graph is an illustrative way to describe the task map. The following diagrams describe each of the task types, and their dependencies.
To describe a regular task, draw a box as in the figure below:

To describe a duplicable task, draw a shadowed box, as in the figure below:

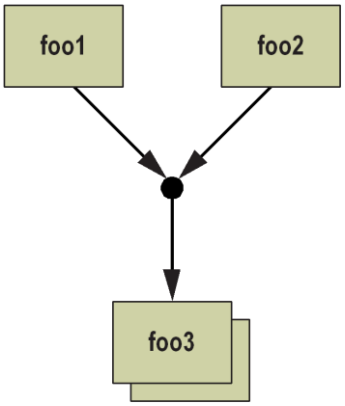
To describe a dependency between two tasks, draw an arrow from the terminating task to the task which is enabled by its termination, as in the figure below:



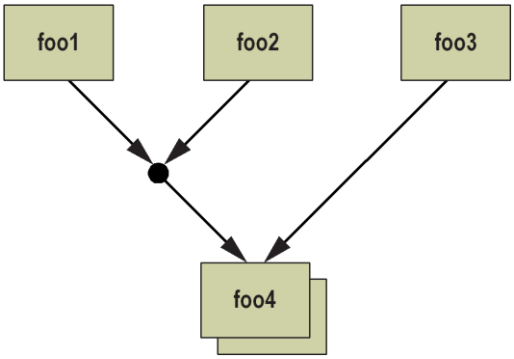
To describe an *AND* connection, draw arrows from the enabling tasks to the task enabled by their termination, as in the figure below:



To describe an *OR* connection, in other words a task that is dependent upon termination of either one of two tasks, draw a dot between the arrows representing enabling events, as shown in the figure below:



To describe a task that is dependent upon termination of either one of two tasks and another task, draw the dependency relationships as shown in figure below:



Variable Token

A token passed by a regular task differs from other token types because it is a variable token, which can have any of three values.

- *True token*—has the value of 1. Software uses a special termination command to send this value. When a task sends a true token, only the tasks that are dependent on the true token are affected.
- *False token*—has the value of 0. Software uses a special termination command to send this value. When a task sends a false token, only the tasks that are dependent on the false token are affected. This is the default token if the software developer does not explicitly specify a different termination token.
- *Don't Care token*—indicates task termination without indicating an exit value.

Duplicable tasks and dummy tasks send a simple termination token to their dependent tasks.

Quotas

A task's quota is the number of instances of a duplicable task available for allocation to cores. The CSU passes a unique instance number to each core executing an instance of a duplicable task, which replaces the loop counter used in a conventional uniprocessor programming model. For example, assume that we wish to execute 17 instances of task X in parallel. Task X will have a quota of 17. The HyperCore scheduler will allocate instances of task X to the available cores with unique instance numbers from 0 to 16.

The quota is different from the number of cores executing the task. The quota can be much larger than the number of cores available in a particular implementation, in which case the hardware automatically breaks up the workload into groups of task instances (called *partial packs*), as shown in Figure 26. Only if the number of cores exceeds or equal to the number of task instances and the cores are not busy executing other tasks, it is guaranteed that the task instances will execute simultaneously. Software must ensure that task instances do not try to update each other's memory, because there is no synchronization between task instances except at termination, when the last task instance finishes execution and sends its termination token.

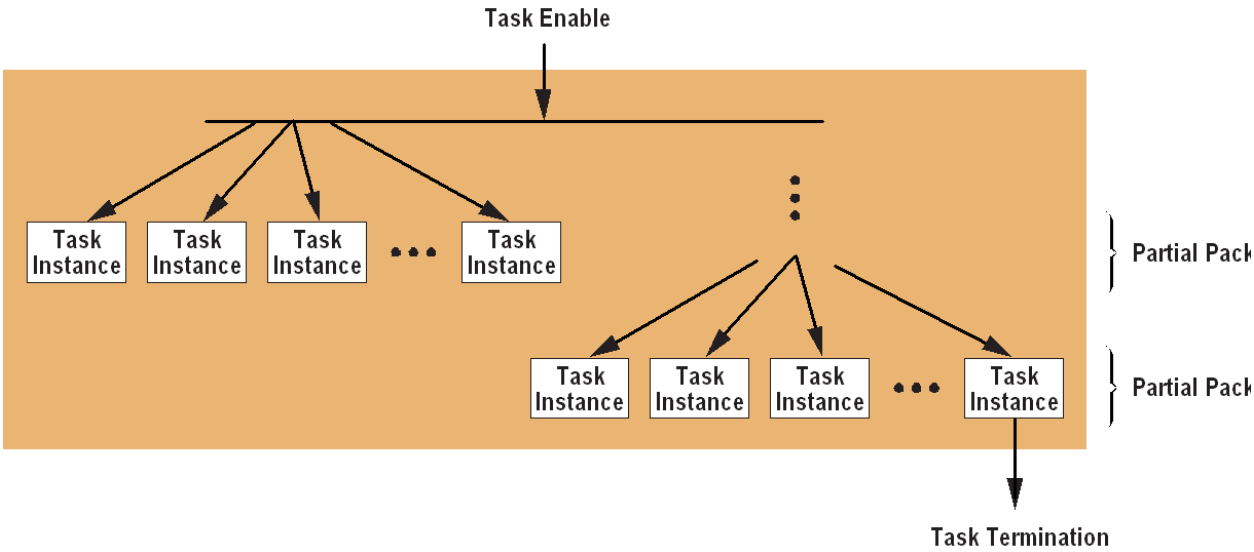


Figure 23 - Execution of partial packs

In order to set the quota we use a Macro command HAL_SET_QUOTA(task_name, N) as in the example below:

```
#define N 17

/* The init task initializes the quota of task evaluate */
void init (void)
{
        HAL_SET_QUOTA(evaluate, N);
```

```
}
```

These lines of code would set a quota of 17 instances for the duplicable task “evaluate”.

The `HAL_TASK_INST` macro returns the instance number of the current instance of the `evaluate` task. This can, for example, substitute the loop counter variable in the serial version of the algorithm.

Making a Task Map

The first step toward making task map is to draw its dependency graph, as shown in Figure 7 for this simple example.

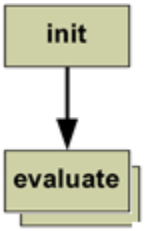


Figure 24 - Dependency Graph for the Simple Example

The next step is to write a source file describing the graph. The task map file contains a simple, intuitive language for describing a dependency graph, as shown below. The development toolchain analyze this file and create an object file capable of programming the scheduler to contain the task map.

```
regular task init ()
duplicable task evaluate (init/u)
```

The part of a task declaration statement in parentheses “()” is the enabling condition. In this case, `init` is a pre-enabled task and `evaluate` is dependent on termination of `init`. The `u` indicates a “do not care” dependency on the termination value returned by `init`.

In summary, when the `init` task starts running, it initializes the numbers of instances (quota) of the `evaluate` task, then it terminates. After `init` terminates, the `evaluate` task is executed. Each available core executes one instance of that task, with a unique instance number for each task instance. A total of 17 instances of `evaluate` task are executed (each on different core), in each instance the value of `HAL_TASK_INST` macro would hold the instance number.

Task Declaration Syntax

Syntax	Behavior
regular task foo()	Declares *foo* as a regular task
duplicable task foo2 ()	Declares *foo2* as a duplicable task
dummy task foo3 ()	Declares *foo3* as a dummy task

Task-Enabling Condition Syntax

The code inside the () in the table above is the task-enabling condition. A task is usually enabled for execution by the termination of other tasks. For regular task, a Boolean value is passed when they terminate which can be used as a condition for enabling other tasks to execute. Examples of the task condition syntax are shown in the table below:

Syntax	Behavior
(foo/0)	Enabled when regular task *foo* passes a false token
(foo/1)	Enabled when regular task *foo* passes a true token
(foo/u)	Enabled when regular task *foo* terminates, without regard to the value of any token passed by *foo*
(foo)	Enabled when non-regular task *foo* terminates
(foo | bar)	Enabled after task *foo* or task *bar* terminate
(foo & bar)	Enabled after task *foo* and task *bar* terminate
(foo & (bar | boo))	Enabled after task foo and task bad or task boo terminate

Each regular task must specify the condition next to its name in the condition syntax. Any task which is declared without a condition is a pre-enabled task, which is enabled for execution immediately after the reset task (initialization task 0) terminates. Duplicable and dummy tasks do not provide termination values, so it is an error to declare a task-enabling condition that tests the termination value of these task types.

In order for a regular task to pass a true/false token, so following macros are used HAL_TASK_RET_TRUE returns true from regular task and HAL_TASK_RET_FALSE returns false from regular task.

שאלות הכנה לחלק א'

1. מדוע התעורר הצורך במעבד מרובה ליבות? מדוע התכנון של מעבד סידרתי לא יוכל לספק שיפור ביצועים?

2. מבנה מעבד ה-HAL:
   א. לאיזה קטגוריה בחלוקה של Flynn משתייך מעבד ה-HAL ?
   ב. לאיזו ארכיטקטורה משתייך מעבד ה-HAL ?
   ג. באילו רמות מיקבול נעשה שימוש במעבד ה-HAL ? (תיתכן יותר מרמת מיקבול אחת)
   ד. אילו בעיות קיימות בזכרון משותף בין מעבדים רבים? איזה פתרון ניתן לבעיות אלו במעבד ה-HAL ? [if8]

3. נתון אלגוריתם אשר מכיל 10% אשר אינם ניתנים למיקבול:
   א. חשב את גורם ההאצה עבור מעבד בעל 2,4,8,16,32,64,128,256 ואינסוף ליבות.
   ב. הצג גרף.
   ג. החל מאיזה כמות ליבות, הוספת ליבות נוספות הופכת זניחה ביחס להאצה?

4. סוגי משימות.
   א. אילו סוגי משימות קיימים במודל ה-TOP של Plurality ?
   ב. בקובץ task.map כתובה השורה הבאה:

```
duplicable task f(func/u)
```

מהו סוג המשימה func ?

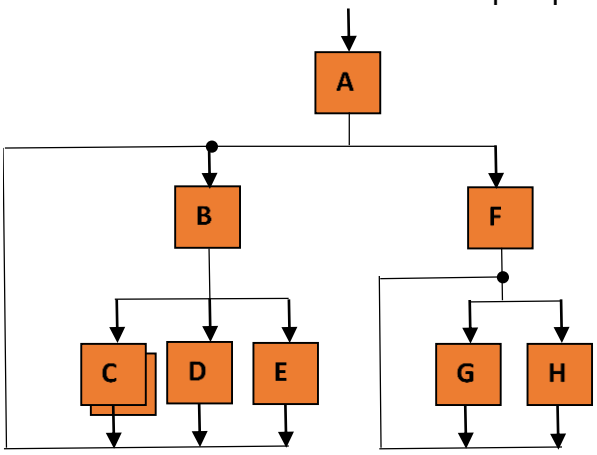
   ג. מדוע לדעתכם לא ניתן לבנות תנאי המבוסס על ערך החזר של משימה duplicable ?

5. נתון קובץ task.map הבא:

```
regular task init()
regular task a(init/u)
regular task b(init/u)
dummy task c(a & b)
duplicable task d(c & e)
regular task e(init/u)
```

צייר את הגרף המתאים.

6. נתון הגרף התלויות הבא:



כתבו קובץ task.map מתאים.

7.

א. אילו 3 סוגי לולאות קיימות?

ב. כיצד נוכל לממש לולאות אלו, כך שתוכן הלולאה ירוץ באופן מקבילי.

ג. ציירו את גרפי התלויות עבור סעיף ב' וכתבו את קבצי ה-map המתאימים. מה יהיה התוכן של כל פונקציה?

Load Balancing

In the HAL processor load balancing is managed by the Synchronizer/Scheduler, which allocates tasks and monitors cores; it knows when a task terminates and when a core becomes available for another task. Figure 28 shows an execution sequence that might occur using a task map like the one in seen in question 6 in preparation questions for Part 1. Task-allocation time is very fast—typically 5 to 9 cycles (Figure 11)—so many fine-grained tasks can be executed in parallel, taking full advantage of many cores.

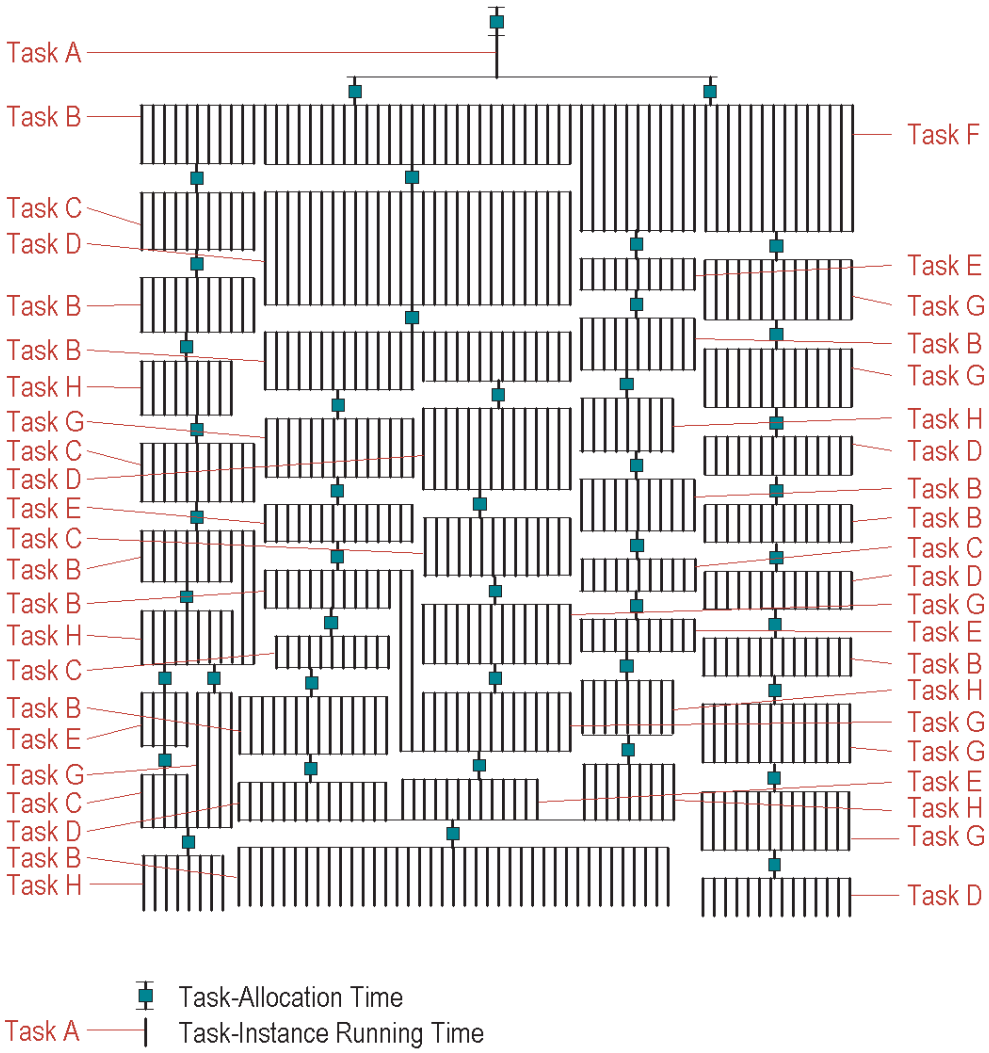


Figure 25 - Load Balancing

The Synchronizer/Scheduler manages task-allocation so that the maximum number of core execution cycles are used productively. Task instances are allocated to sets of cores simultaneously, but each core may complete its task instance at a slightly different time than other cores in that set, due to occasional collisions in the memory-access network or different control flow of tasks.

Each core starts automatically when a task instance is allocated to it. The core continues execution until it encounters an instruction signifying the end of the task. The core then halts, informs the Synchronizer/Scheduler that it has completed, and waits to receive another task instance.

When a core completes execution of a task instance, it immediately becomes free to execute another instance of the same task (irrespective of whether other cores in that set have completed their execution) or of any other task. When all instances of a task terminate, the task itself is terminated, and this enables any dependent tasks that are waiting for its termination. In this way, available task instances are dynamically allocated to available cores.

The hardware Synchronizer/Scheduler is capable of managing hundreds of tasks and cores in parallel. The maximum number of tasks that can be allocated by the Synchronizer/Scheduler or terminated by the cores in one cycle is equal to the total number of cores. There is no static mapping of tasks to cores; any core can run any task. Cores always run tasks and task instances to completion; they never suspend one task to switch to another task.