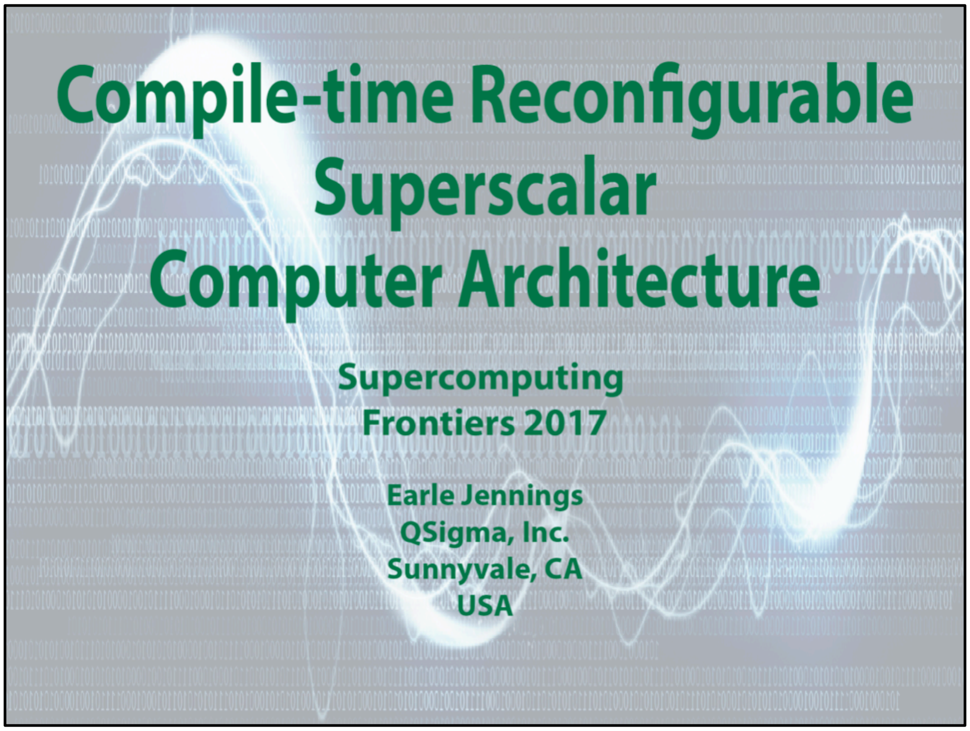
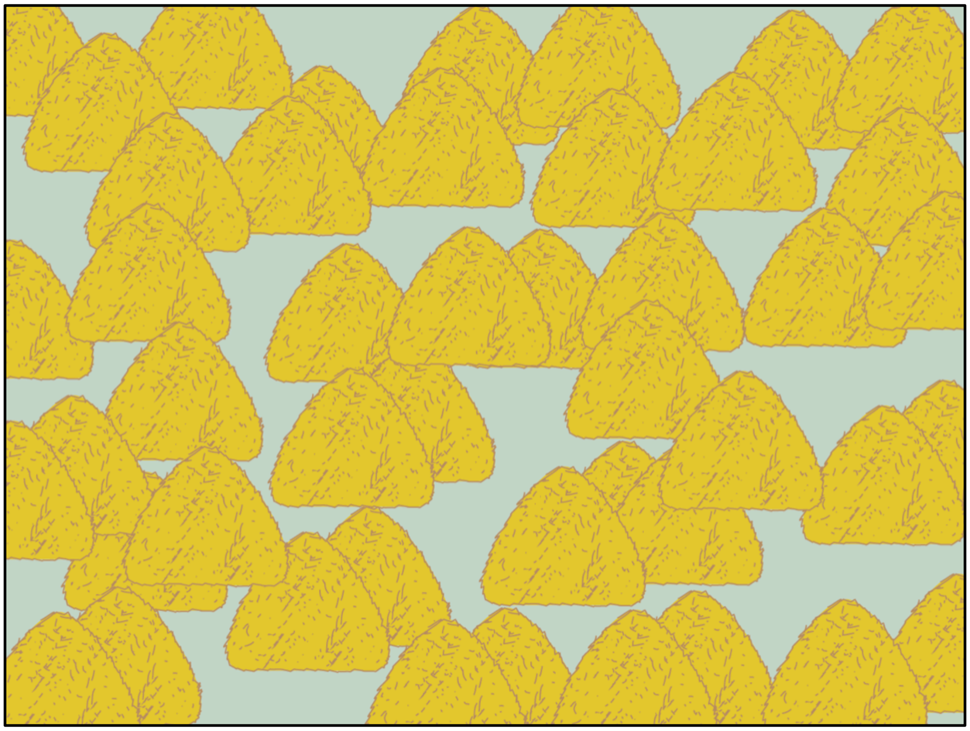


Compile-time Reconfigurable Superscalar Computer Architecture

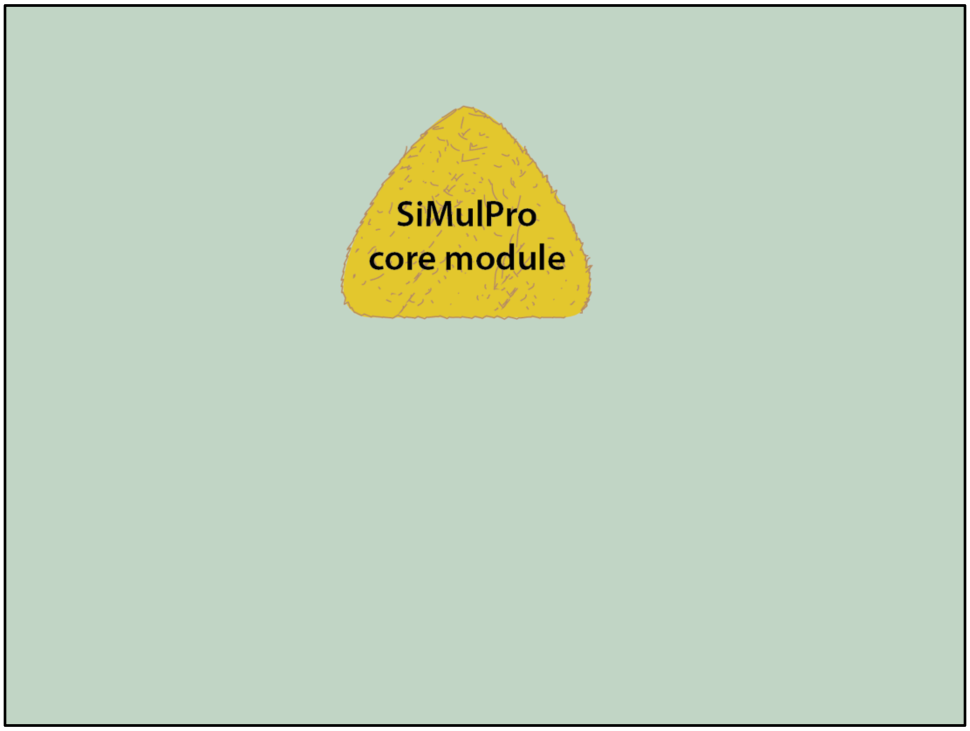
Supercomputing Frontiers 2017

Earle Jennings
QSigma, Inc.
Sunnyvale, CA
USA

Hello! I’m Earle Jennings. This presentation introduces a fundamentally new computer architecture in terms of:
Application compatibility with an existing superscalar microprocessor,
Minimization of energy use, and
Optimized local Sparse Matrix Manipulation

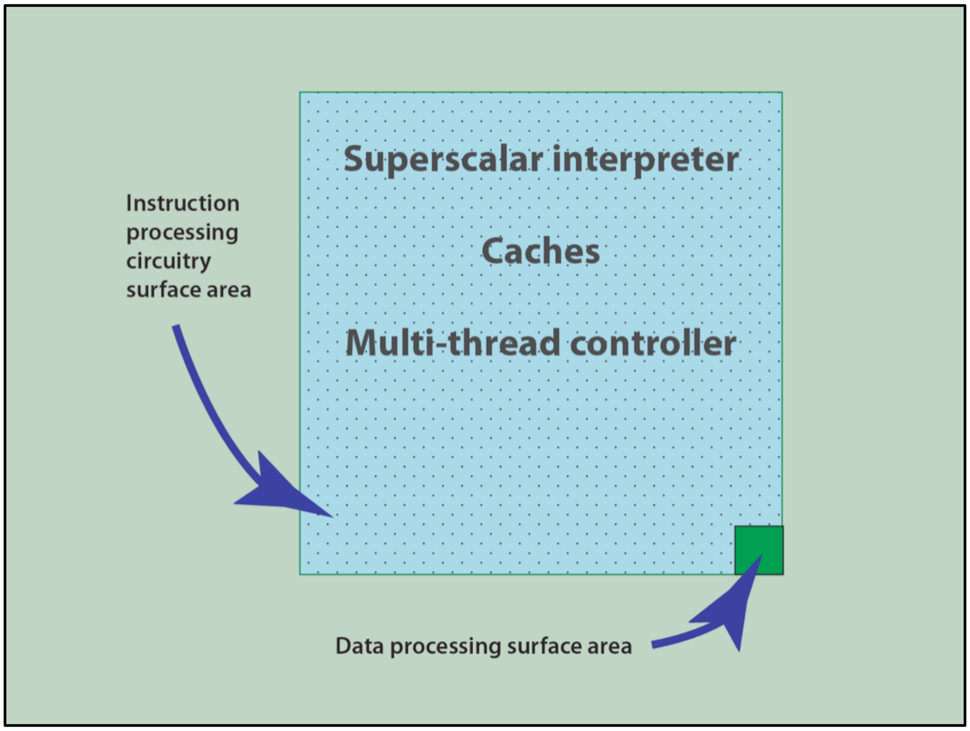


QSigma began researching computer architectures in 2001.
Gradually our search narrowed to a single hay stack.

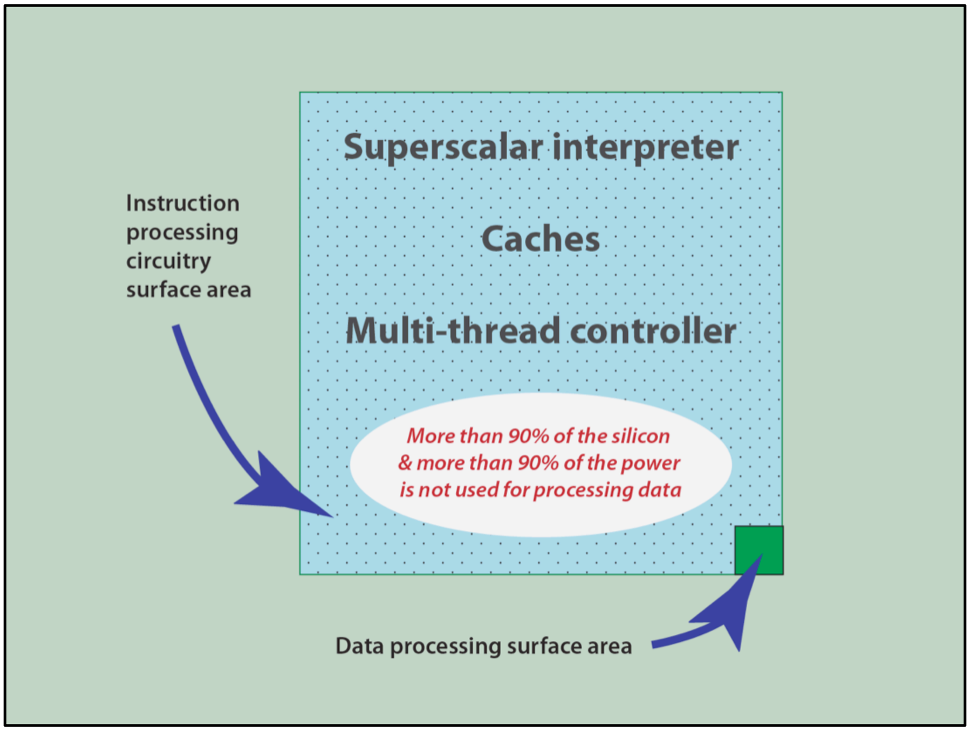


SiMulPro
core module

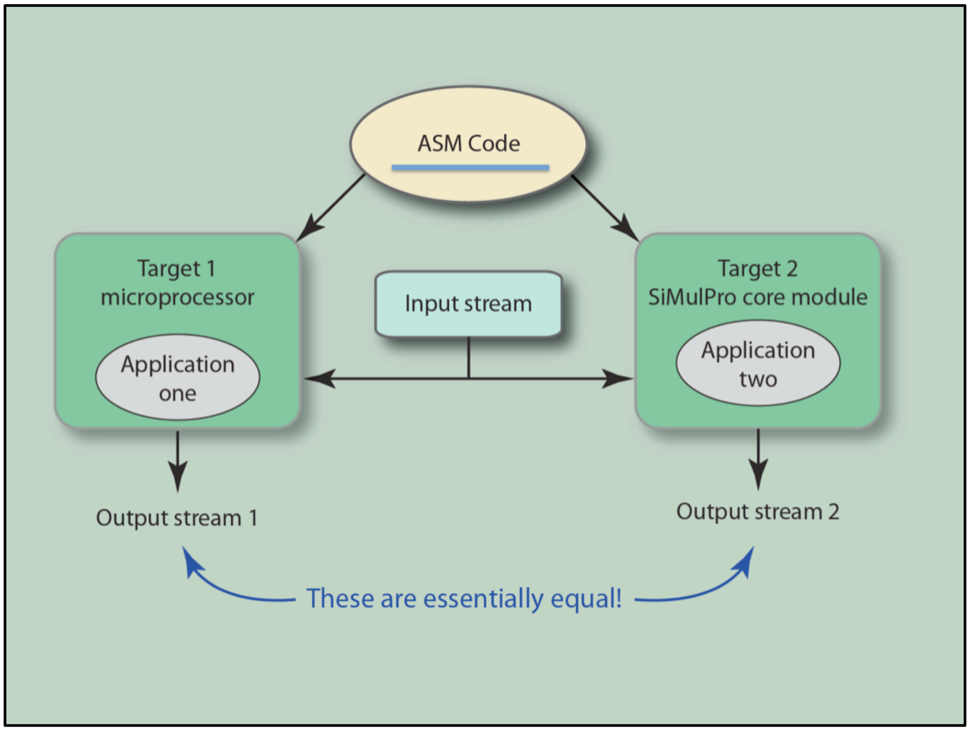
This is called a Simultaneous Multi-Processor - SiMulPro core module.
Application compatibility, results from semantic compatibility, with the microprocessor.
The superscalar instruction interpreter and multi-thread controller are converted into software tools.
Instruction caching disappears through an innovation in VLIW instruction processing.
Sparse matrix operations, and neural network modeling, are locally performed in the core modules and require no data caching.


Superscalar interpreter
Caches
Multi-thread controller
Instruction processing circuitry surface area
Data processing surface area

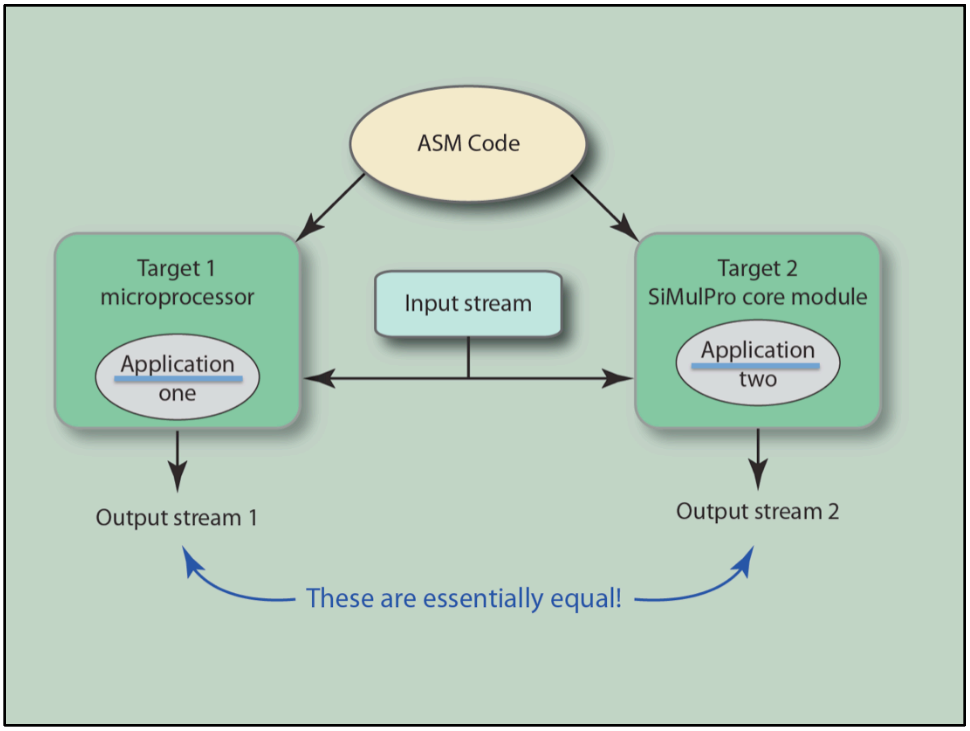
Today’s high performance, superscalar microprocessor, includes all of the elements of this frame.
The blue box, includes the superscalar instruction interpreter, the caches, and the multi-thread controller.


Superscalar interpreter
Caches
Multi-thread controller
Instruction processing circuitry surface area
More than 90% of the silicon & more than 90% of the power is not used for processing data
Data processing surface area

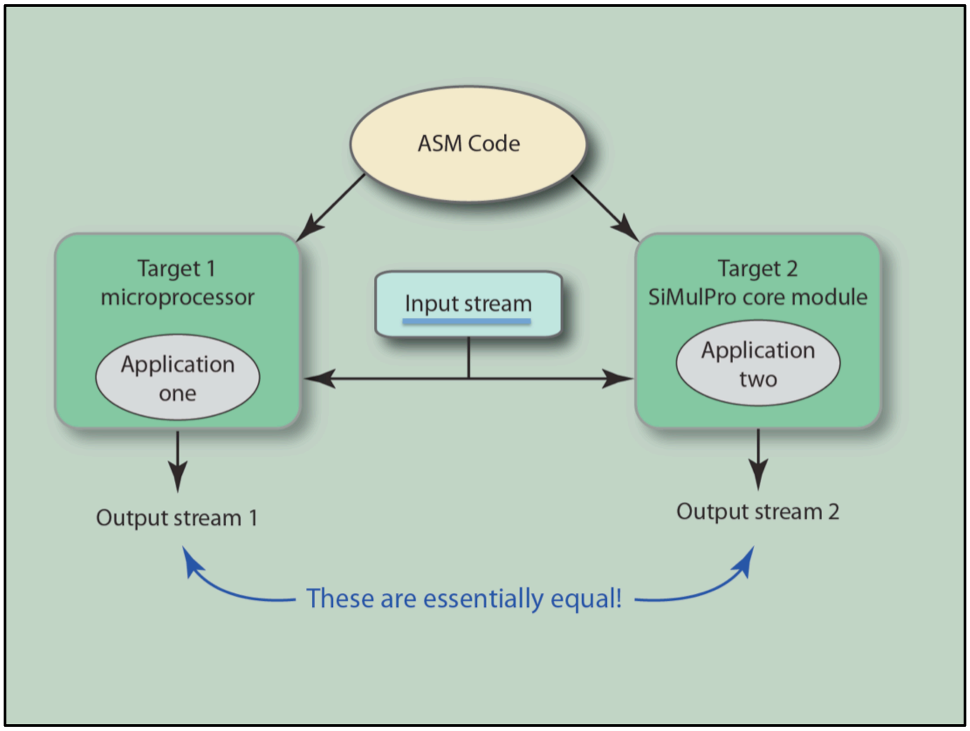
Only the green box with the data processing resources, actually processes the data!
It is a small part of the silicon and power usage!
Removing the blue box components provides, at least, a 10X reduction in **surface area and energy consumption.**
In order to remove that overhead, while remaining application compatible, a new processor must be semantically compatible with the existing microprocessor.



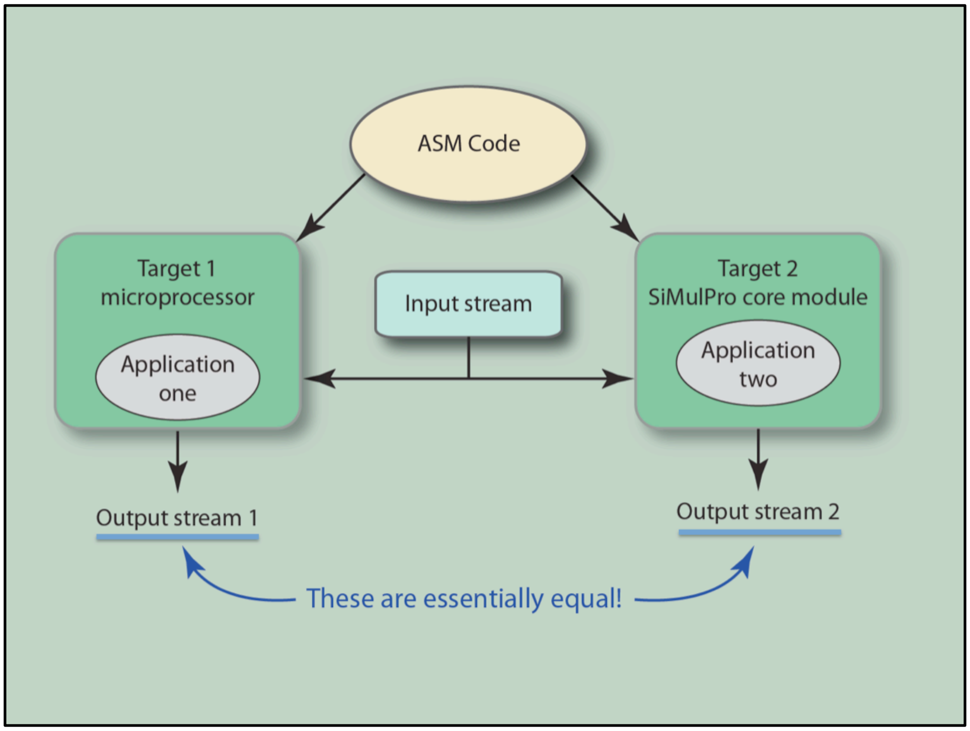
This is an example of semantic compatibility.
Each assembly code program



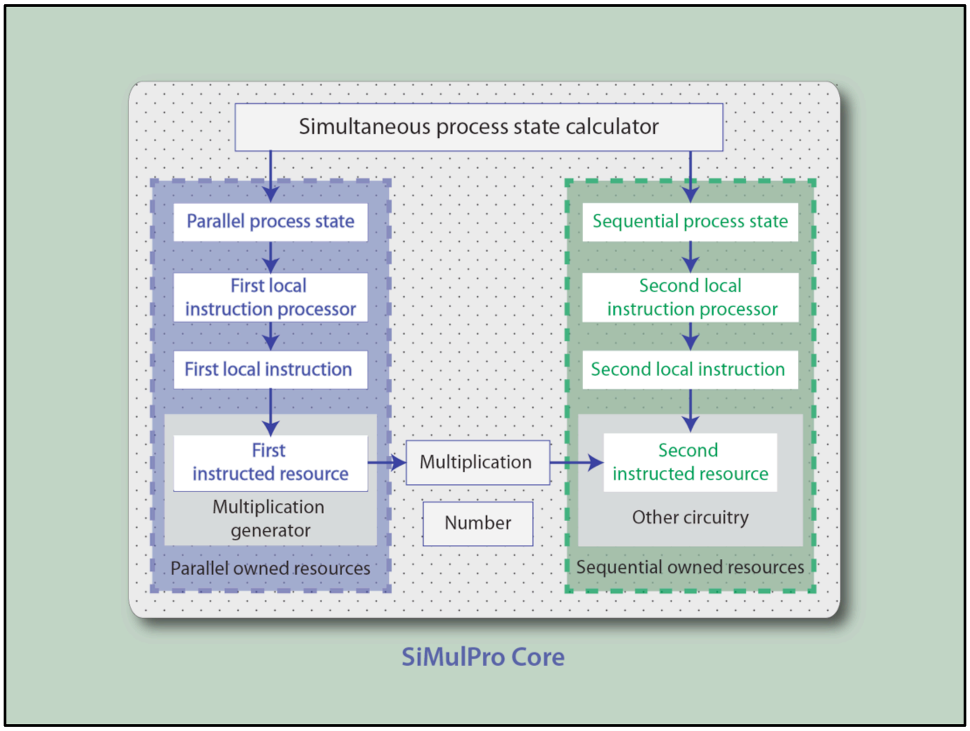
Generates an application in each these targets



Which, when stimulated by the same input stream



generate essentially the same output streams.



Simultaneous process state calculator

Parallel process state

First local instruction processor

First local instruction

First instructed resource

Multiplication generator

Parallel owned resources

Multiplication

Number

Sequential process state

Second local instruction processor

Second local instruction

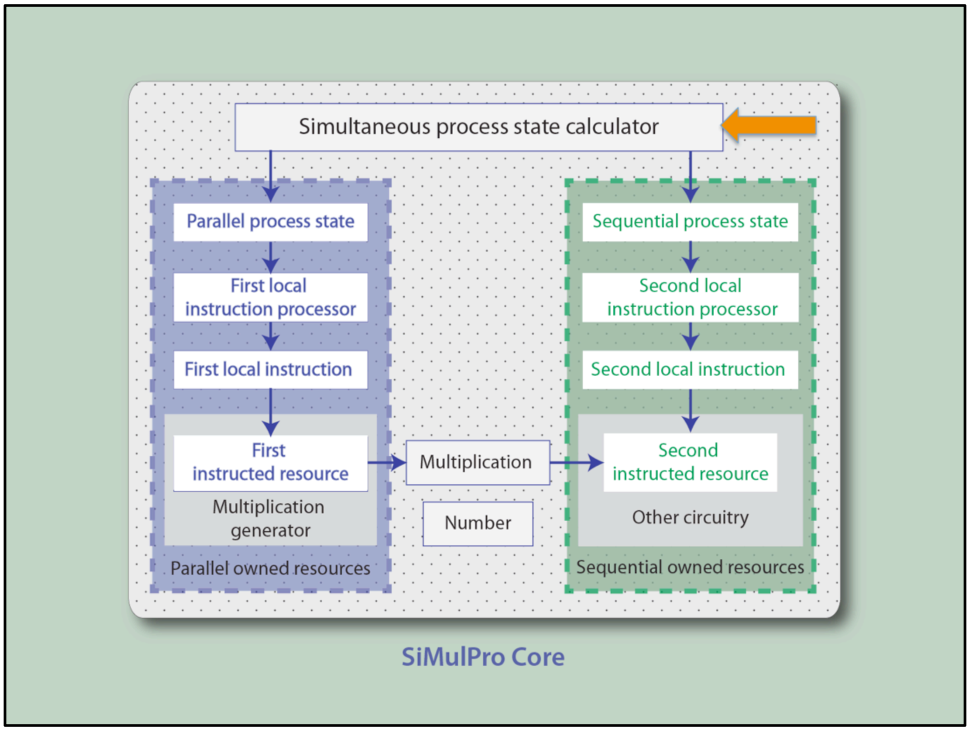
Second instructed resource

Other circuitry

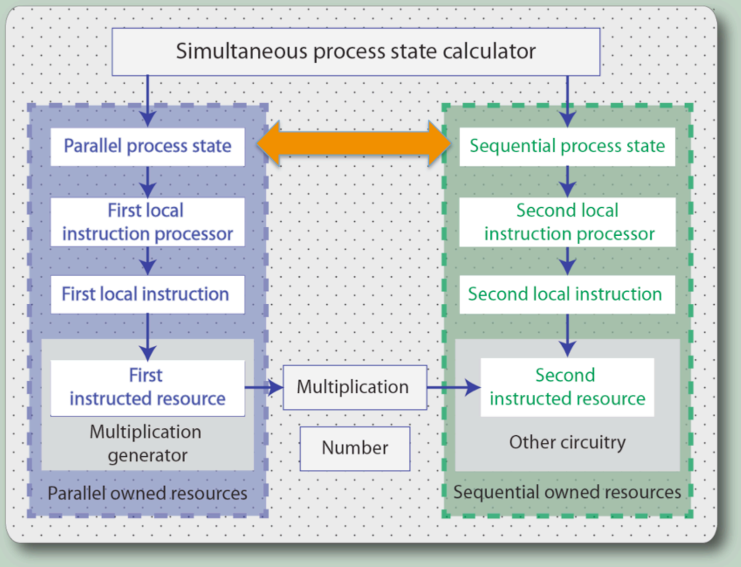
Sequential owned resources

SiMulPro Core

Here is a very simple SiMulPro Core

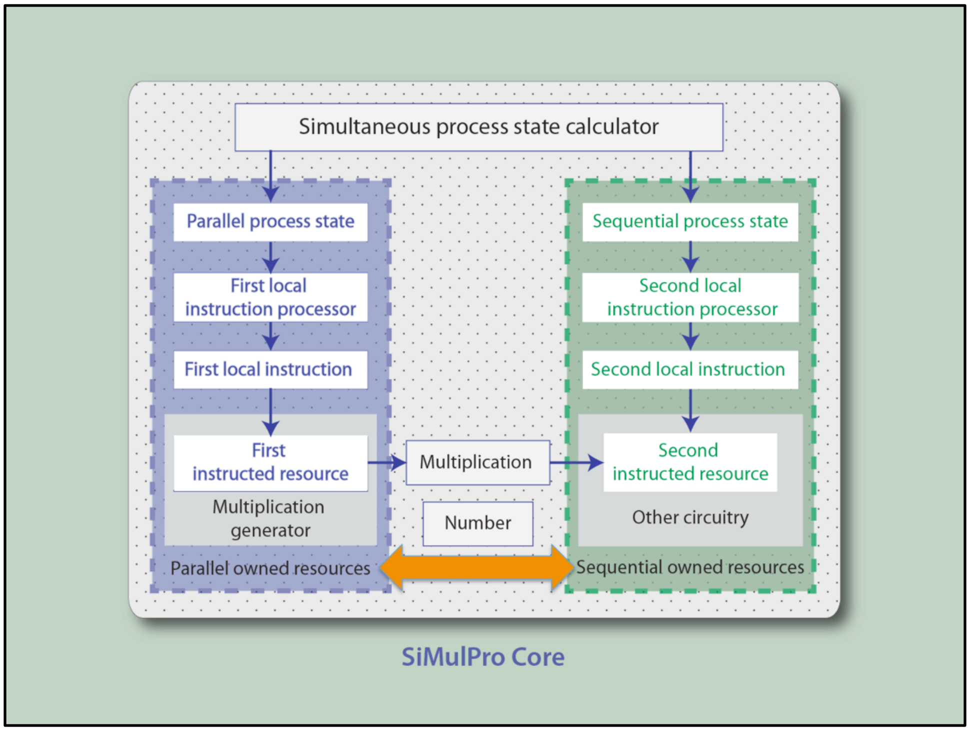


Which implements a process state calculator

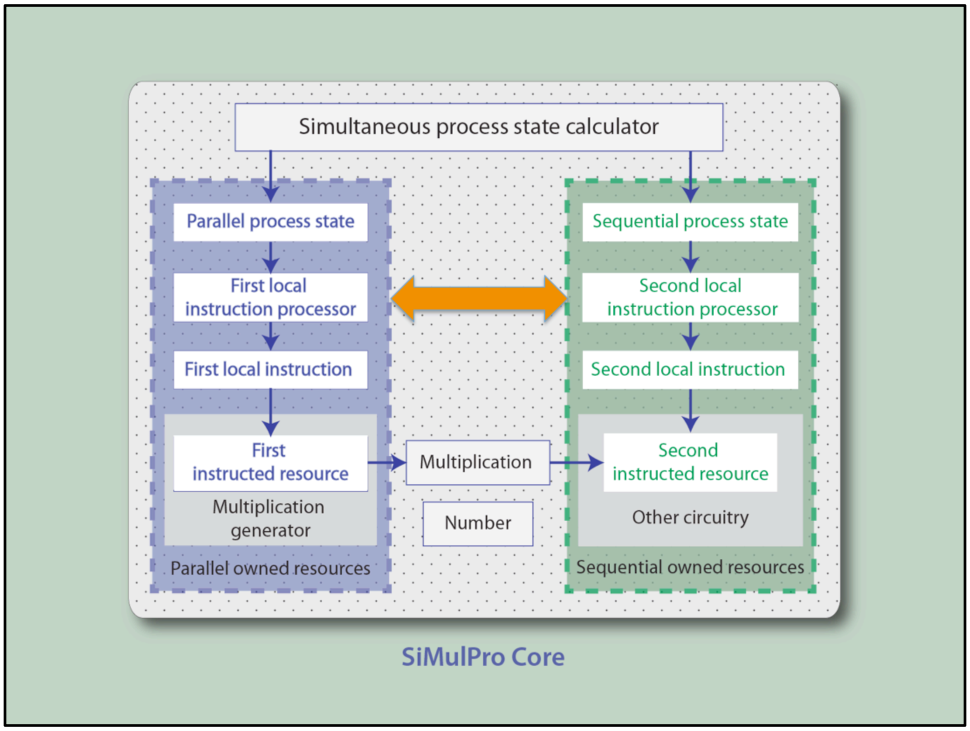

Simultaneous process state calculator
Parallel process state
Sequential process state
First local instruction processor
Second local instruction processor
First local instruction
Second local instruction
First instructed resource
Multiplication
Second instructed resource
Multiplication generator
Number
Other circuitry
Parallel owned resources
Sequential owned resources

SiMulPro Core

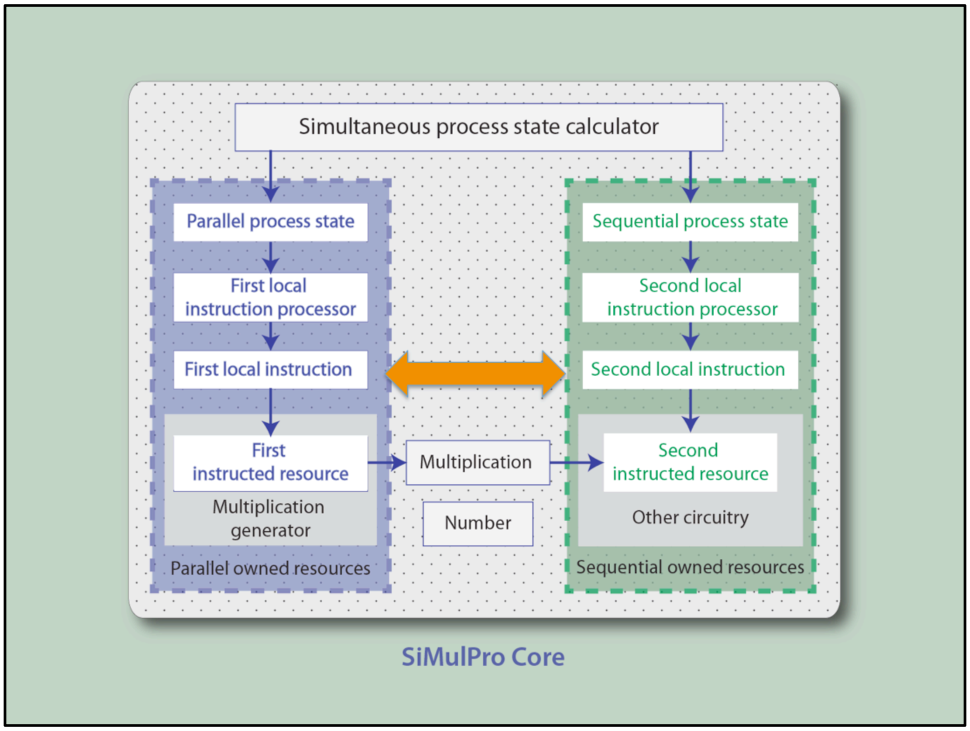
issuing process states for executing these processes.



Each simultaneous process, separately owns the instructed resources of the core.
Each instructed resource, includes a local instruction processor.



The state of the owning process stimulates each owned, local instruction processor.
Each instruction processor accesses local instruction memory for its resource



Simultaneous process state calculator

Parallel process state

First local instruction processor

First local instruction

First instructed resource

Multiplication generator

Parallel owned resources

Multiplication

Number

Sequential process state

Second local instruction processor

Second local instruction

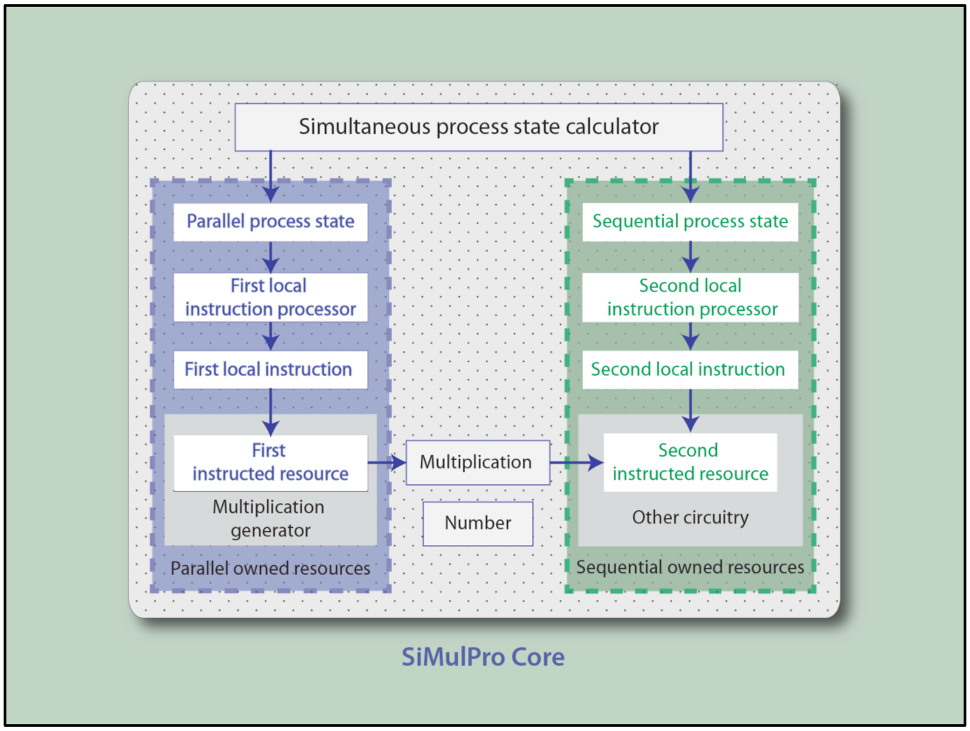
Second instructed resource

Other circuitry

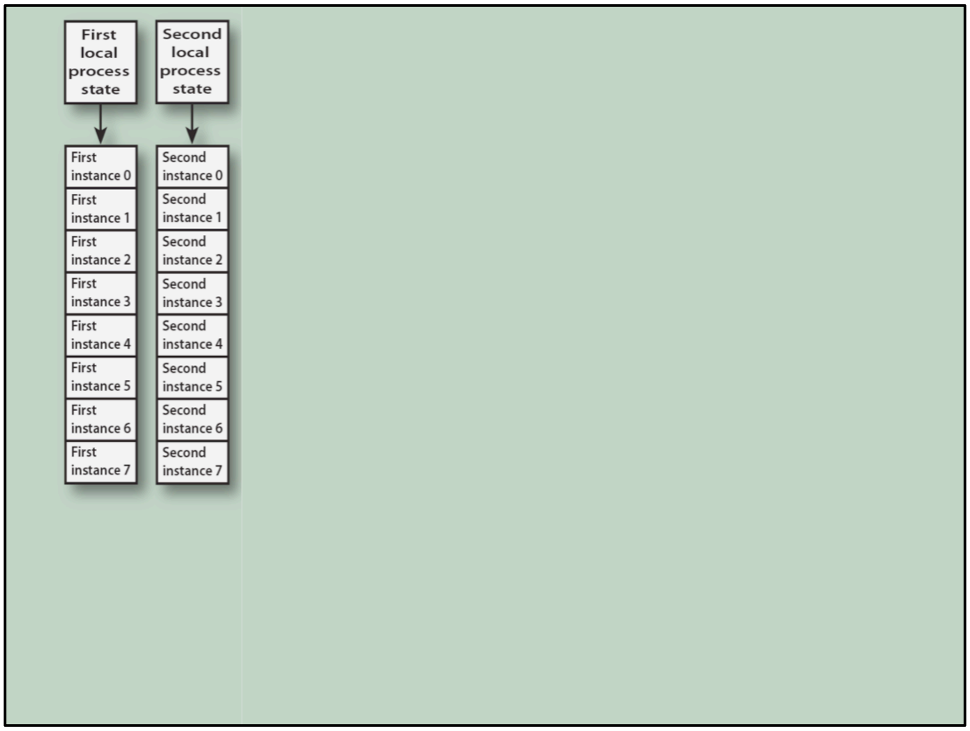
Sequential owned resources

SiMulPro Core

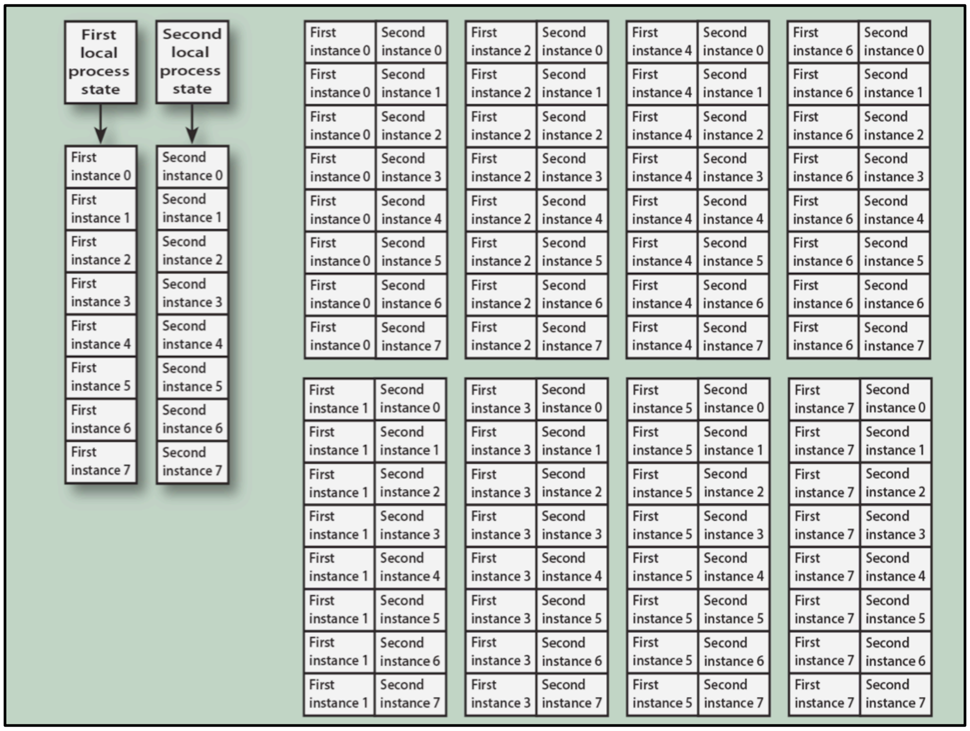
To generate a local instruction which directs the resource.



Each process owns separate instructed resources, like an adder, so that the processes do not stall each other.



Here is an example use of this SiMulPro core.
Assume that the first and second processes each have a range of 8 process states (or local instructions).



First local process state

First instance 0
First instance 1
First instance 2
First instance 3
First instance 4
First instance 5
First instance 6
First instance 7

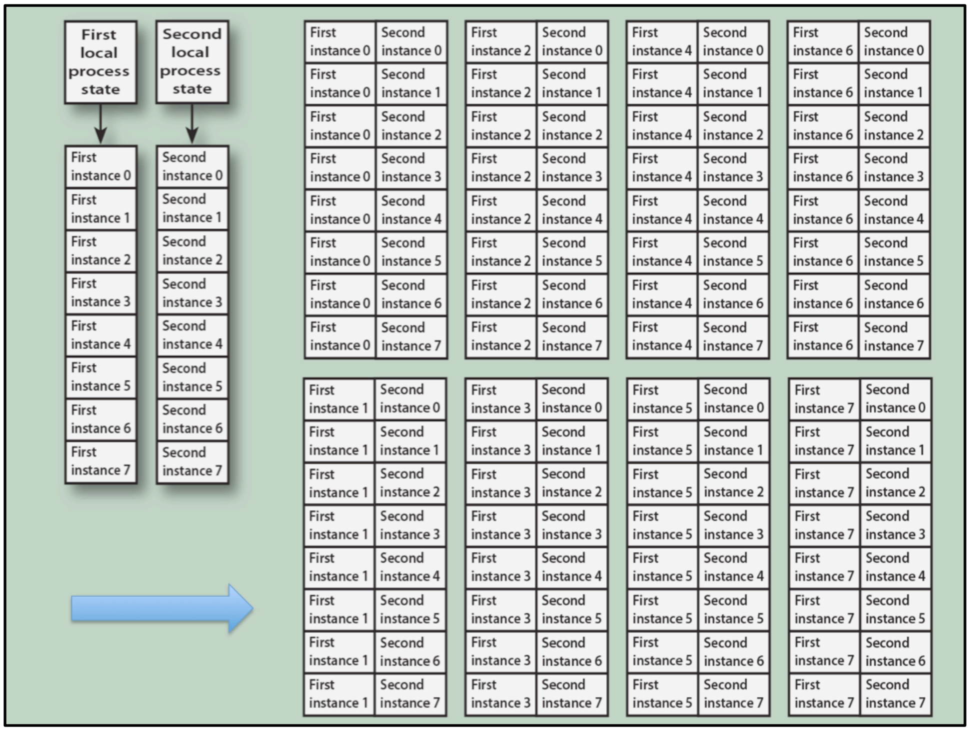
Second local process state

Second instance 0
Second instance 1
Second instance 2
Second instance 3
Second instance 4
Second instance 5
Second instance 6
Second instance 7

First instance 0	Second instance 0	First instance 2	Second instance 0	First instance 4	Second instance 0	First instance 6	Second instance 0
First instance 0	Second instance 1	First instance 2	Second instance 1	First instance 4	Second instance 1	First instance 6	Second instance 1
First instance 0	Second instance 2	First instance 2	Second instance 2	First instance 4	Second instance 2	First instance 6	Second instance 2
First instance 0	Second instance 3	First instance 2	Second instance 3	First instance 4	Second instance 3	First instance 6	Second instance 3
First instance 0	Second instance 4	First instance 2	Second instance 4	First instance 4	Second instance 4	First instance 6	Second instance 4
First instance 0	Second instance 5	First instance 2	Second instance 5	First instance 4	Second instance 5	First instance 6	Second instance 5
First instance 0	Second instance 6	First instance 2	Second instance 6	First instance 4	Second instance 6	First instance 6	Second instance 6
First instance 0	Second instance 7	First instance 2	Second instance 7	First instance 4	Second instance 7	First instance 6	Second instance 7

First instance 1	Second instance 0	First instance 3	Second instance 0	First instance 5	Second instance 0	First instance 7	Second instance 0
First instance 1	Second instance 1	First instance 3	Second instance 1	First instance 5	Second instance 1	First instance 7	Second instance 1
First instance 1	Second instance 2	First instance 3	Second instance 2	First instance 5	Second instance 2	First instance 7	Second instance 2
First instance 1	Second instance 3	First instance 3	Second instance 3	First instance 5	Second instance 3	First instance 7	Second instance 3
First instance 1	Second instance 4	First instance 3	Second instance 4	First instance 5	Second instance 4	First instance 7	Second instance 4
First instance 1	Second instance 5	First instance 3	Second instance 5	First instance 5	Second instance 5	First instance 7	Second instance 5
First instance 1	Second instance 6	First instance 3	Second instance 6	First instance 5	Second instance 6	First instance 7	Second instance 6
First instance 1	Second instance 7	First instance 3	Second instance 7	First instance 5	Second instance 7	First instance 7	Second instance 7

A VLIW memory, supporting these same independent operations, requires a much larger memory, of 64 VLIW instructions.



First local process state

First instance 0
First instance 1
First instance 2
First instance 3
First instance 4
First instance 5
First instance 6
First instance 7

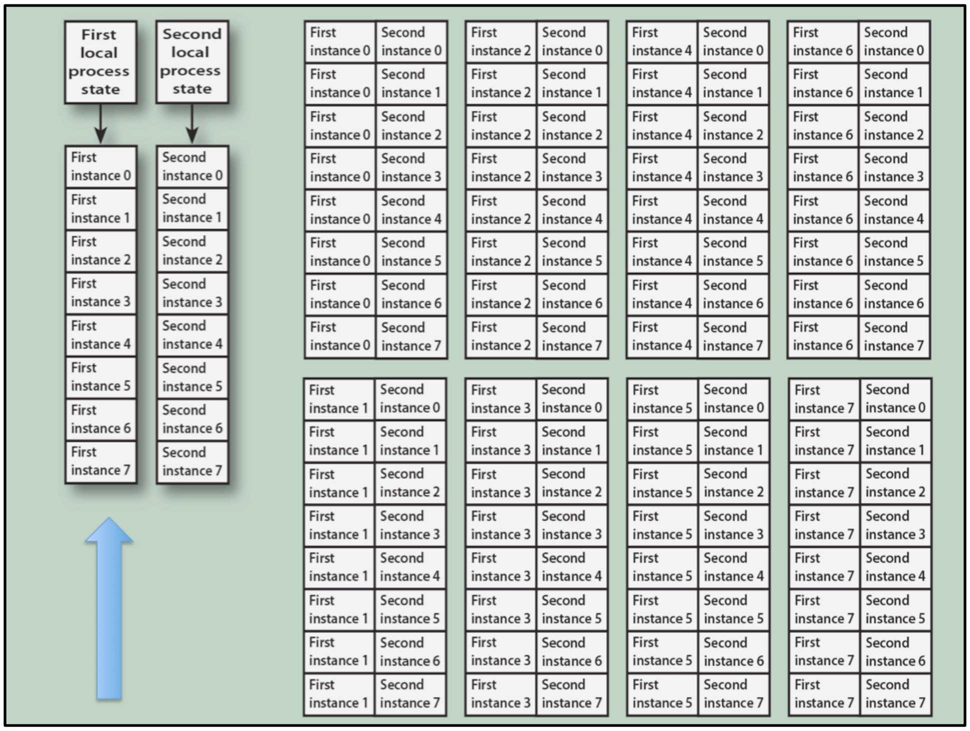
Second local process state

Second instance 0
Second instance 1
Second instance 2
Second instance 3
Second instance 4
Second instance 5
Second instance 6
Second instance 7

First instance 0	Second instance 0	First instance 2	Second instance 0	First instance 4	Second instance 0	First instance 6	Second instance 0
First instance 0	Second instance 1	First instance 2	Second instance 1	First instance 4	Second instance 1	First instance 6	Second instance 1
First instance 0	Second instance 2	First instance 2	Second instance 2	First instance 4	Second instance 2	First instance 6	Second instance 2
First instance 0	Second instance 3	First instance 2	Second instance 3	First instance 4	Second instance 3	First instance 6	Second instance 3
First instance 0	Second instance 4	First instance 2	Second instance 4	First instance 4	Second instance 4	First instance 6	Second instance 4
First instance 0	Second instance 5	First instance 2	Second instance 5	First instance 4	Second instance 5	First instance 6	Second instance 5
First instance 0	Second instance 6	First instance 2	Second instance 6	First instance 4	Second instance 6	First instance 6	Second instance 6
First instance 0	Second instance 7	First instance 2	Second instance 7	First instance 4	Second instance 7	First instance 6	Second instance 7

First instance 1	Second instance 0	First instance 3	Second instance 0	First instance 5	Second instance 0	First instance 7	Second instance 0
First instance 1	Second instance 1	First instance 3	Second instance 1	First instance 5	Second instance 1	First instance 7	Second instance 1
First instance 1	Second instance 2	First instance 3	Second instance 2	First instance 5	Second instance 2	First instance 7	Second instance 2
First instance 1	Second instance 3	First instance 3	Second instance 3	First instance 5	Second instance 3	First instance 7	Second instance 3
First instance 1	Second instance 4	First instance 3	Second instance 4	First instance 5	Second instance 4	First instance 7	Second instance 4
First instance 1	Second instance 5	First instance 3	Second instance 5	First instance 5	Second instance 5	First instance 7	Second instance 5
First instance 1	Second instance 6	First instance 3	Second instance 6	First instance 5	Second instance 6	First instance 7	Second instance 6
First instance 1	Second instance 7	First instance 3	Second instance 7	First instance 5	Second instance 7	First instance 7	Second instance 7

This right hand side is characteristic of the Multiflow, and the EPIC architecture, which led to the I-64 and Itanium.



First local process state

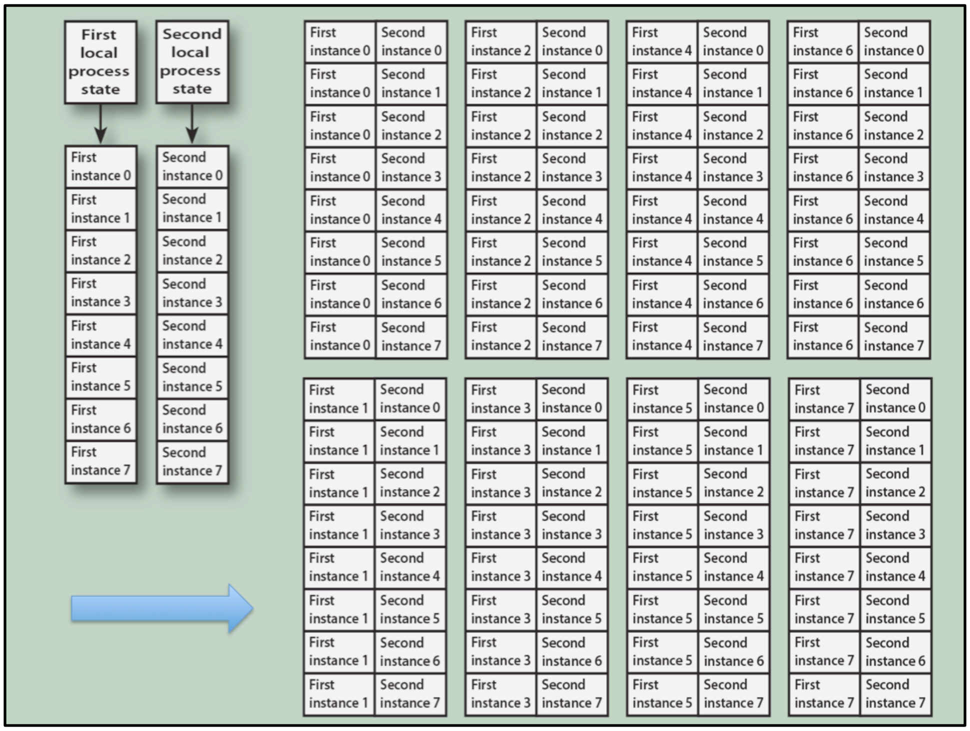
Second local process state

First instance 0	Second instance 0
First instance 1	Second instance 1
First instance 2	Second instance 2
First instance 3	Second instance 3
First instance 4	Second instance 4
First instance 5	Second instance 5
First instance 6	Second instance 6
First instance 7	Second instance 7

First instance 0	Second instance 0	First instance 2	Second instance 0	First instance 4	Second instance 0	First instance 6	Second instance 0
First instance 0	Second instance 1	First instance 2	Second instance 1	First instance 4	Second instance 1	First instance 6	Second instance 1
First instance 0	Second instance 2	First instance 2	Second instance 2	First instance 4	Second instance 2	First instance 6	Second instance 2
First instance 0	Second instance 3	First instance 2	Second instance 3	First instance 4	Second instance 3	First instance 6	Second instance 3
First instance 0	Second instance 4	First instance 2	Second instance 4	First instance 4	Second instance 4	First instance 6	Second instance 4
First instance 0	Second instance 5	First instance 2	Second instance 5	First instance 4	Second instance 5	First instance 6	Second instance 5
First instance 0	Second instance 6	First instance 2	Second instance 6	First instance 4	Second instance 6	First instance 6	Second instance 6
First instance 0	Second instance 7	First instance 2	Second instance 7	First instance 4	Second instance 7	First instance 6	Second instance 7

First instance 1	Second instance 0	First instance 3	Second instance 0	First instance 5	Second instance 0	First instance 7	Second instance 0
First instance 1	Second instance 1	First instance 3	Second instance 1	First instance 5	Second instance 1	First instance 7	Second instance 1
First instance 1	Second instance 2	First instance 3	Second instance 2	First instance 5	Second instance 2	First instance 7	Second instance 2
First instance 1	Second instance 3	First instance 3	Second instance 3	First instance 5	Second instance 3	First instance 7	Second instance 3
First instance 1	Second instance 4	First instance 3	Second instance 4	First instance 5	Second instance 4	First instance 7	Second instance 4
First instance 1	Second instance 5	First instance 3	Second instance 5	First instance 5	Second instance 5	First instance 7	Second instance 5
First instance 1	Second instance 6	First instance 3	Second instance 6	First instance 5	Second instance 6	First instance 7	Second instance 6
First instance 1	Second instance 7	First instance 3	Second instance 7	First instance 5	Second instance 7	First instance 7	Second instance 7

The Mill computer, shows some resemblance to the left hand side, but has **only two** instruction pointers.
The SiMulPro core, supports factoring algorithms into their natural units, of up to 6, or more, simultaneous processes, which operate based upon data availability.
This virtual VLIW space, removes the need for instruction caching.



First local process state

First instance 0
First instance 1
First instance 2
First instance 3
First instance 4
First instance 5
First instance 6
First instance 7

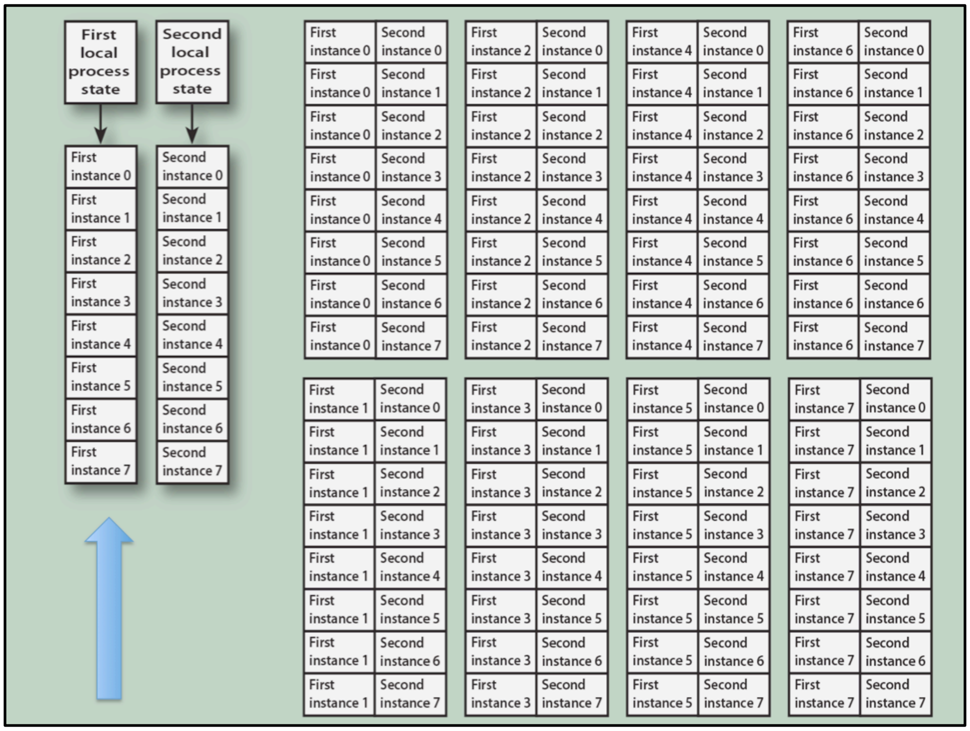
Second local process state

Second instance 0
Second instance 1
Second instance 2
Second instance 3
Second instance 4
Second instance 5
Second instance 6
Second instance 7

First instance 0	Second instance 0	First instance 2	Second instance 0	First instance 4	Second instance 0	First instance 6	Second instance 0
First instance 0	Second instance 1	First instance 2	Second instance 1	First instance 4	Second instance 1	First instance 6	Second instance 1
First instance 0	Second instance 2	First instance 2	Second instance 2	First instance 4	Second instance 2	First instance 6	Second instance 2
First instance 0	Second instance 3	First instance 2	Second instance 3	First instance 4	Second instance 3	First instance 6	Second instance 3
First instance 0	Second instance 4	First instance 2	Second instance 4	First instance 4	Second instance 4	First instance 6	Second instance 4
First instance 0	Second instance 5	First instance 2	Second instance 5	First instance 4	Second instance 5	First instance 6	Second instance 5
First instance 0	Second instance 6	First instance 2	Second instance 6	First instance 4	Second instance 6	First instance 6	Second instance 6
First instance 0	Second instance 7	First instance 2	Second instance 7	First instance 4	Second instance 7	First instance 6	Second instance 7

First instance 1	Second instance 0	First instance 3	Second instance 0	First instance 5	Second instance 0	First instance 7	Second instance 0
First instance 1	Second instance 1	First instance 3	Second instance 1	First instance 5	Second instance 1	First instance 7	Second instance 1
First instance 1	Second instance 2	First instance 3	Second instance 2	First instance 5	Second instance 2	First instance 7	Second instance 2
First instance 1	Second instance 3	First instance 3	Second instance 3	First instance 5	Second instance 3	First instance 7	Second instance 3
First instance 1	Second instance 4	First instance 3	Second instance 4	First instance 5	Second instance 4	First instance 7	Second instance 4
First instance 1	Second instance 5	First instance 3	Second instance 5	First instance 5	Second instance 5	First instance 7	Second instance 5
First instance 1	Second instance 6	First instance 3	Second instance 6	First instance 5	Second instance 6	First instance 7	Second instance 6
First instance 1	Second instance 7	First instance 3	Second instance 7	First instance 5	Second instance 7	First instance 7	Second instance 7

All predecessor VLIW approaches require unique compilers, which negate application compatibility.

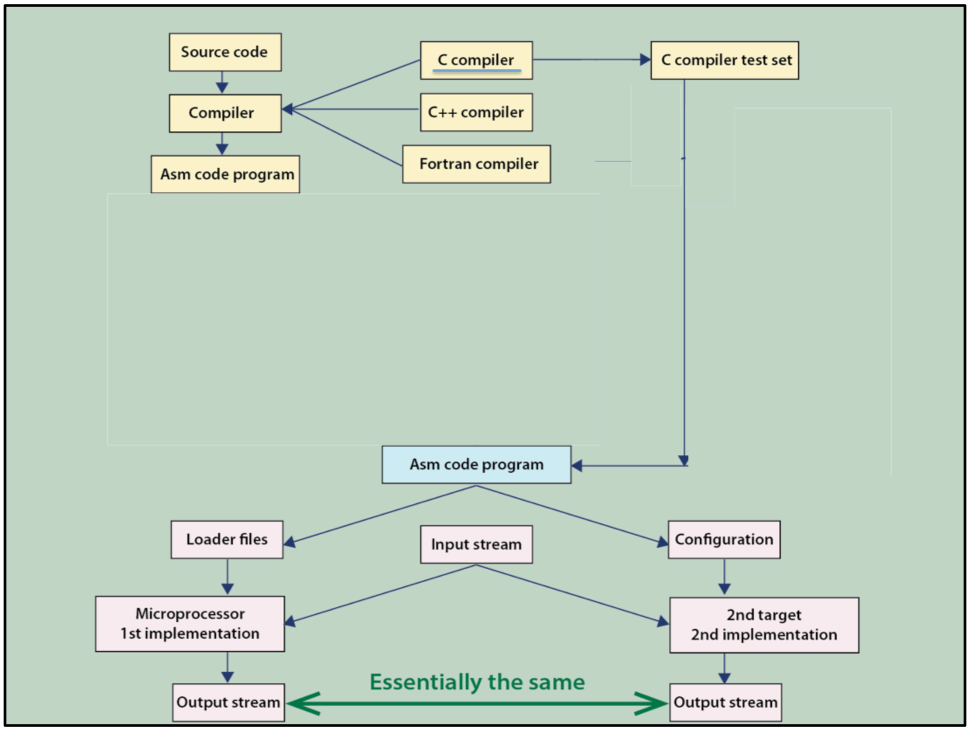


First local process state	Second local process state
First instance 0	Second instance 0
First instance 1	Second instance 1
First instance 2	Second instance 2
First instance 3	Second instance 3
First instance 4	Second instance 4
First instance 5	Second instance 5
First instance 6	Second instance 6
First instance 7	Second instance 7

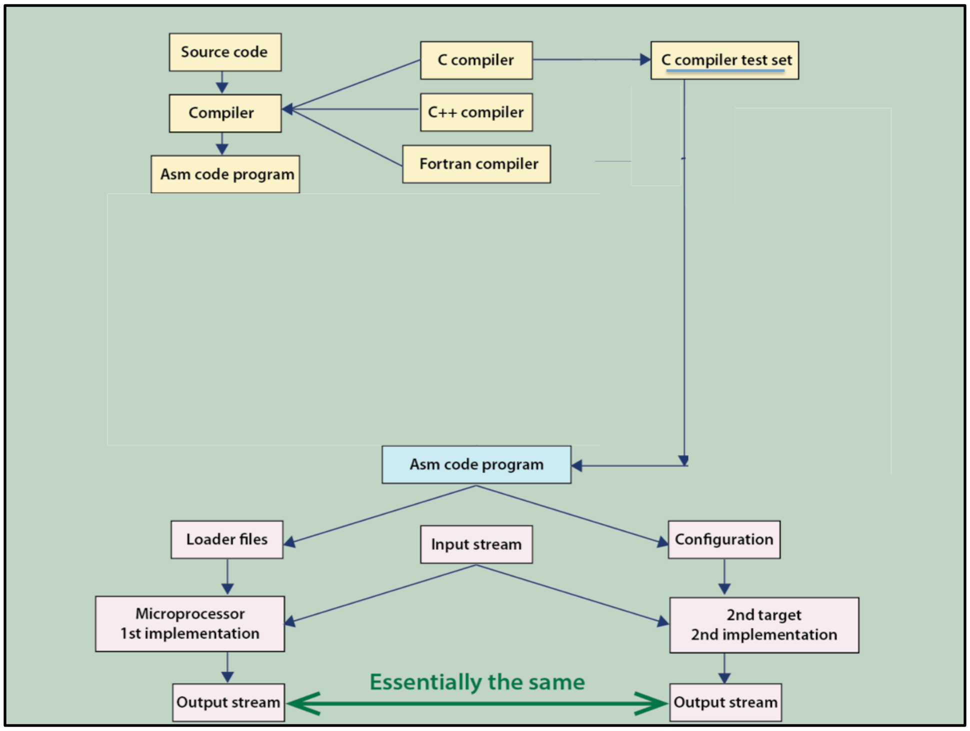
First instance 0	Second instance 0	First instance 2	Second instance 0	First instance 4	Second instance 0	First instance 6	Second instance 0
First instance 0	Second instance 1	First instance 2	Second instance 1	First instance 4	Second instance 1	First instance 6	Second instance 1
First instance 0	Second instance 2	First instance 2	Second instance 2	First instance 4	Second instance 2	First instance 6	Second instance 2
First instance 0	Second instance 3	First instance 2	Second instance 3	First instance 4	Second instance 3	First instance 6	Second instance 3
First instance 0	Second instance 4	First instance 2	Second instance 4	First instance 4	Second instance 4	First instance 6	Second instance 4
First instance 0	Second instance 5	First instance 2	Second instance 5	First instance 4	Second instance 5	First instance 6	Second instance 5
First instance 0	Second instance 6	First instance 2	Second instance 6	First instance 4	Second instance 6	First instance 6	Second instance 6
First instance 0	Second instance 7	First instance 2	Second instance 7	First instance 4	Second instance 7	First instance 6	Second instance 7

First instance 1	Second instance 0	First instance 3	Second instance 0	First instance 5	Second instance 0	First instance 7	Second instance 0
First instance 1	Second instance 1	First instance 3	Second instance 1	First instance 5	Second instance 1	First instance 7	Second instance 1
First instance 1	Second instance 2	First instance 3	Second instance 2	First instance 5	Second instance 2	First instance 7	Second instance 2
First instance 1	Second instance 3	First instance 3	Second instance 3	First instance 5	Second instance 3	First instance 7	Second instance 3
First instance 1	Second instance 4	First instance 3	Second instance 4	First instance 5	Second instance 4	First instance 7	Second instance 4
First instance 1	Second instance 5	First instance 3	Second instance 5	First instance 5	Second instance 5	First instance 7	Second instance 5
First instance 1	Second instance 6	First instance 3	Second instance 6	First instance 5	Second instance 6	First instance 7	Second instance 6
First instance 1	Second instance 7	First instance 3	Second instance 7	First instance 5	Second instance 7	First instance 7	Second instance 7

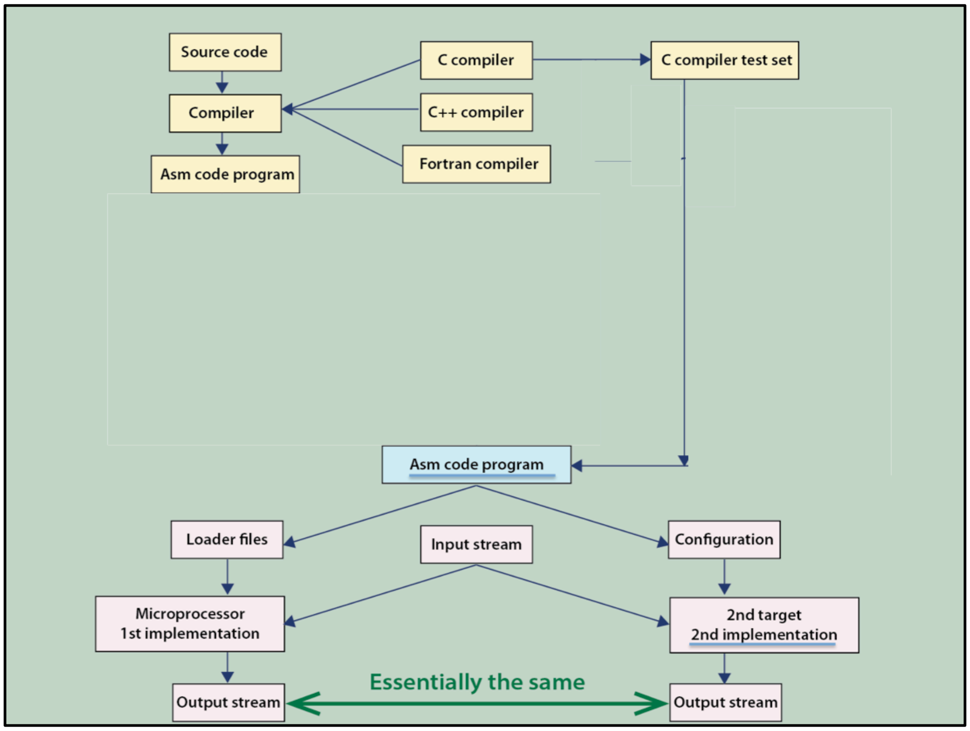
The SiMulPro cores are semantically compatible with the existing microprocessor, sharing established, compiler tools through the assembly code, generation stage.



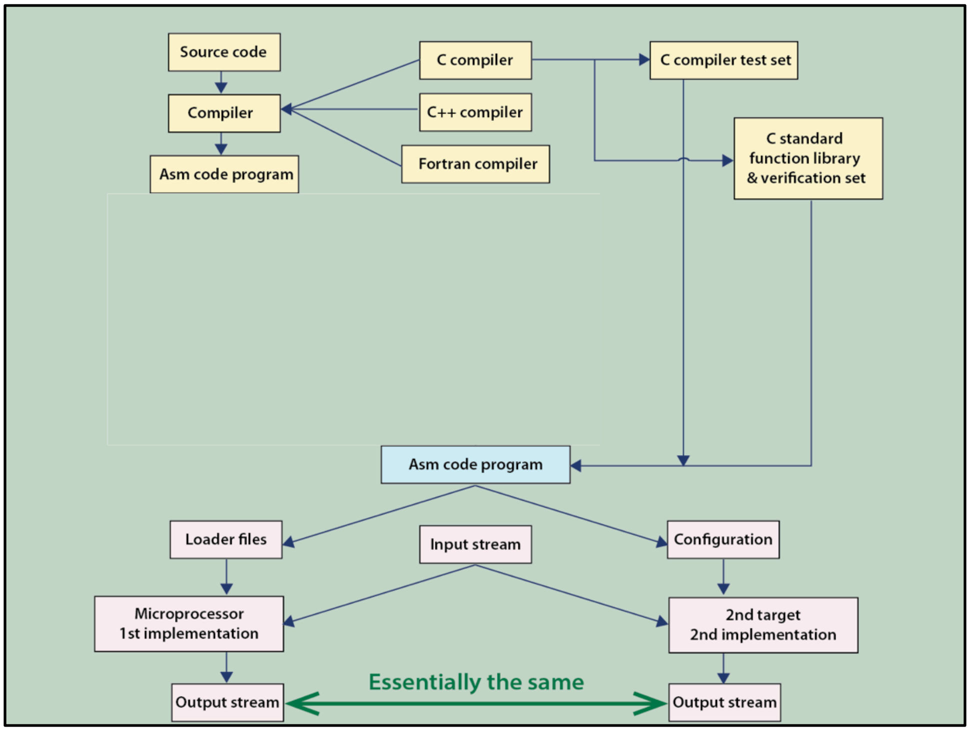
Here is a first step in verifying, compiler compatibility. Let’s look at a C compiler.



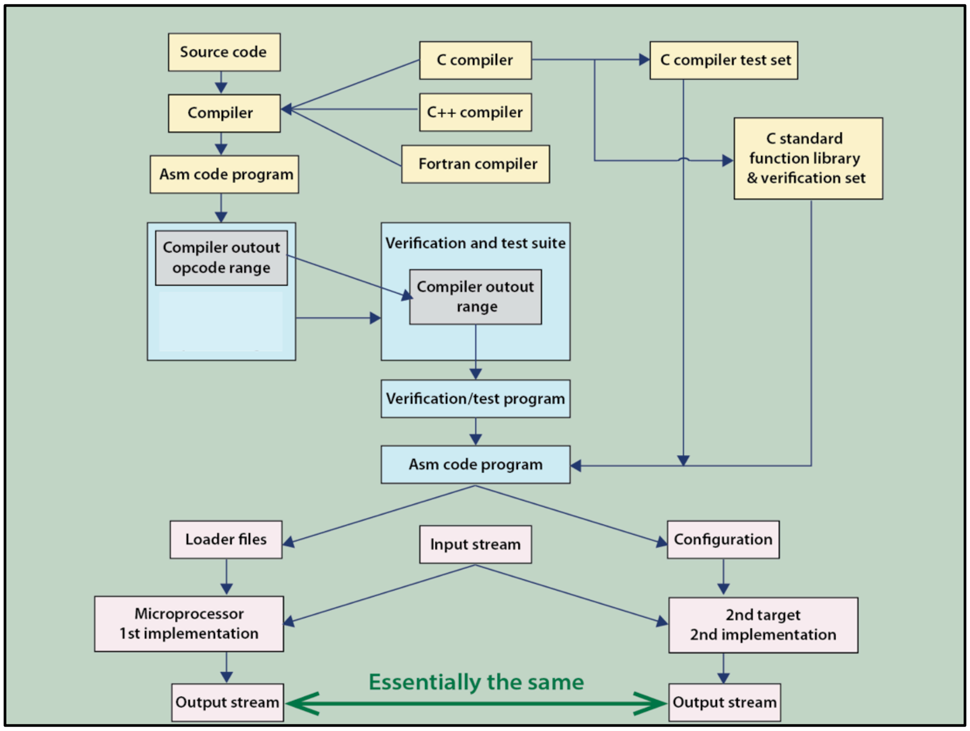
It has a compiler, test set, used to confirm code generation, targeting the microprocessor.
This compiler test set



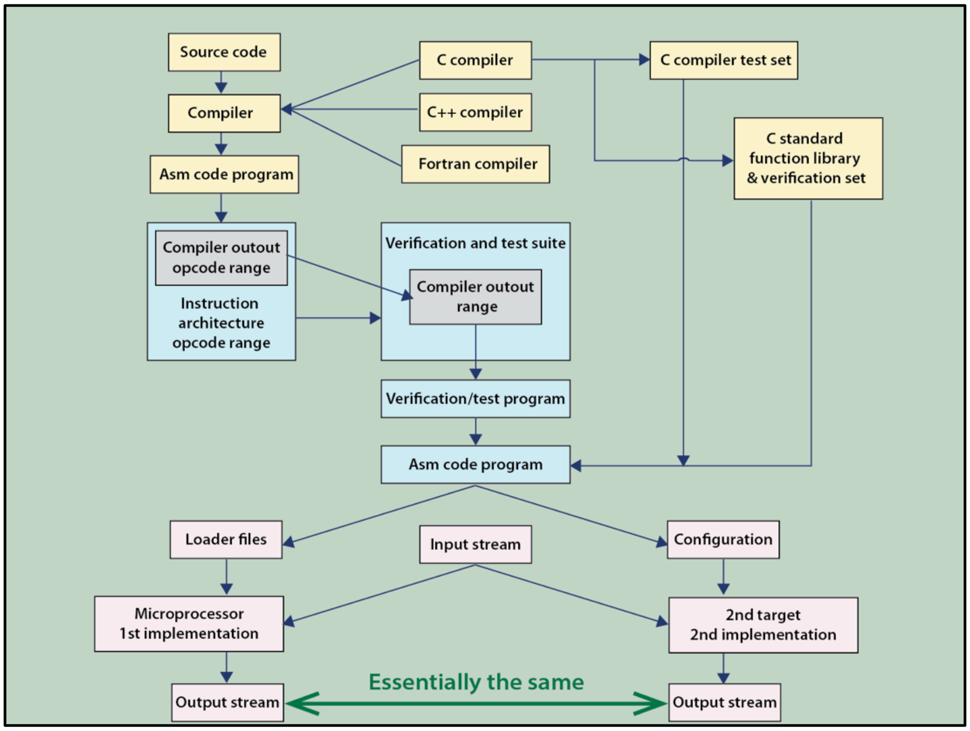
can also be used to confirm the 2nd target from its assembly code programs.



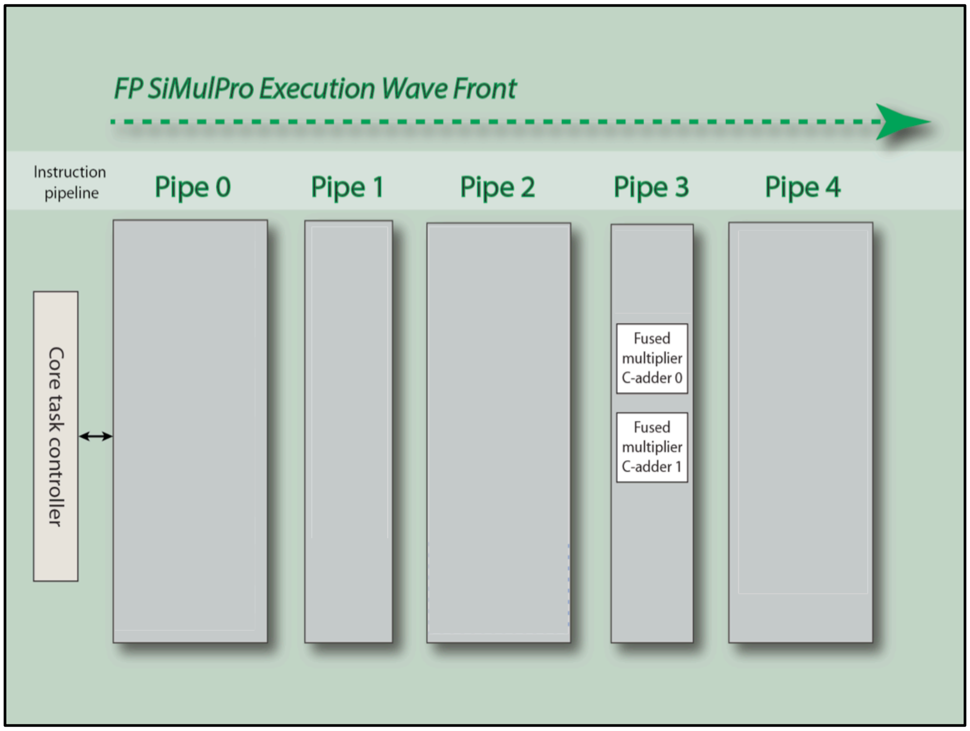
A 2nd step adds C function libraries and verification sets.



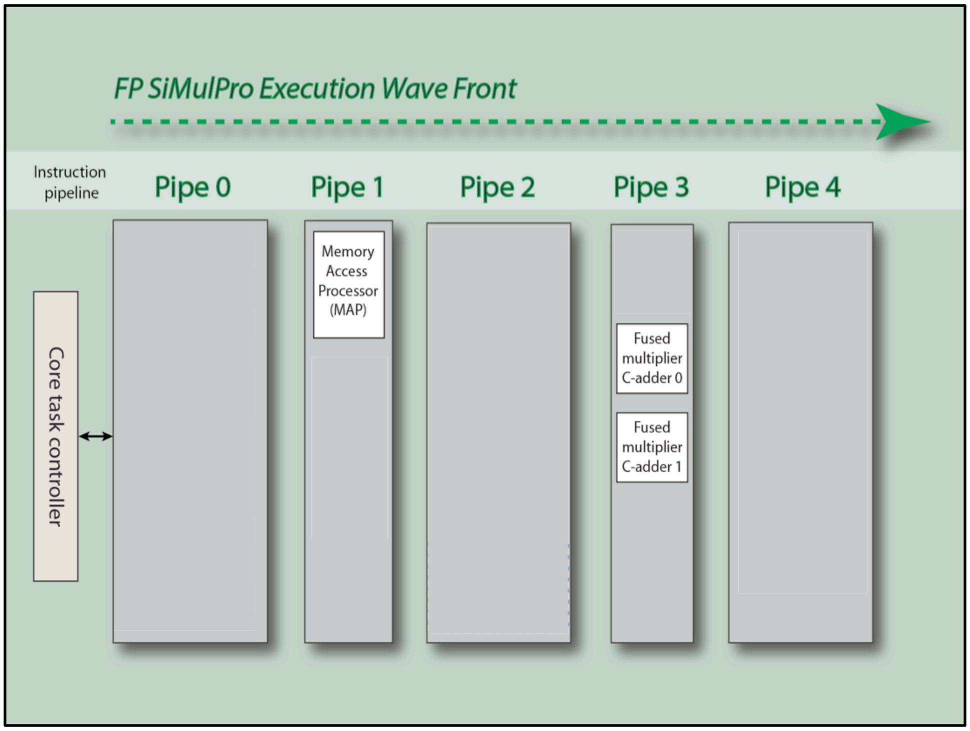
The 3rd step starts from the compiler, output, opcode range


Source code
Compiler
Asm code program
C compiler
C++ compiler
Fortran compiler
C compiler test set
C standard function library & verification set
Compiler outout opcode range
Instruction architecture opcode range
Verification and test suite
Compiler outout range
Verification/test program
Asm code program
Loader files
Input stream
Configuration
Microprocessor 1st implementation
2nd target 2nd implementation
Output stream
Essentially the same
Output stream

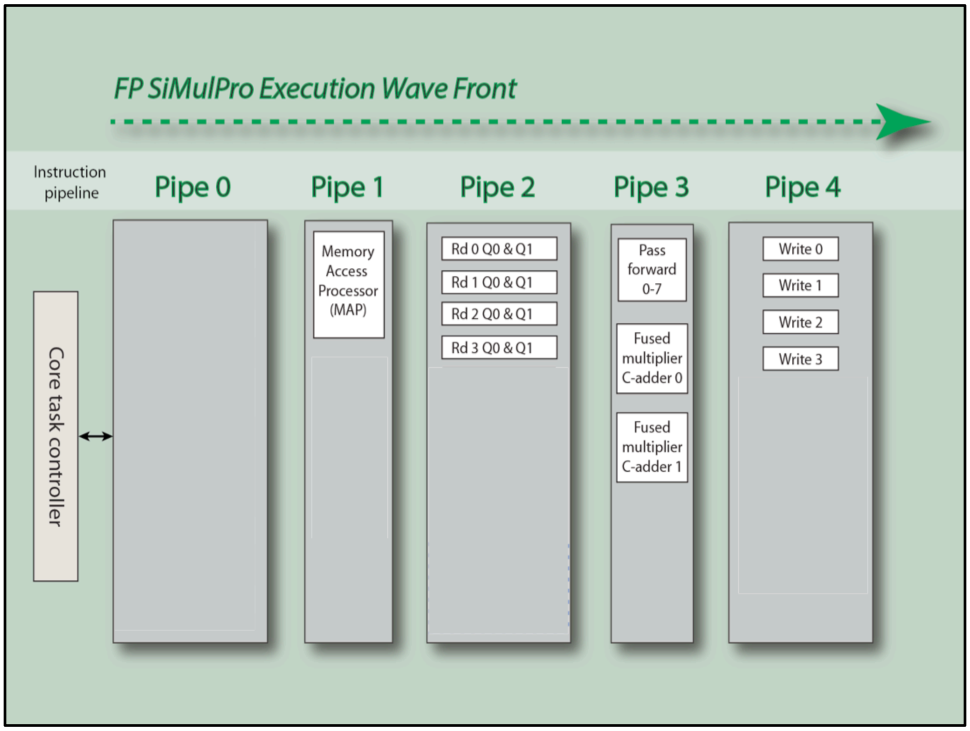
and extends to include more, possibly all, of the instruction set architecture.


FP SiMulPro Execution Wave Front
Instruction pipeline
Pipe 0
Pipe 1
Pipe 2
Pipe 3
Pipe 4
Core task controller
Fused multiplier C-adder 0
Fused multiplier C-adder 1

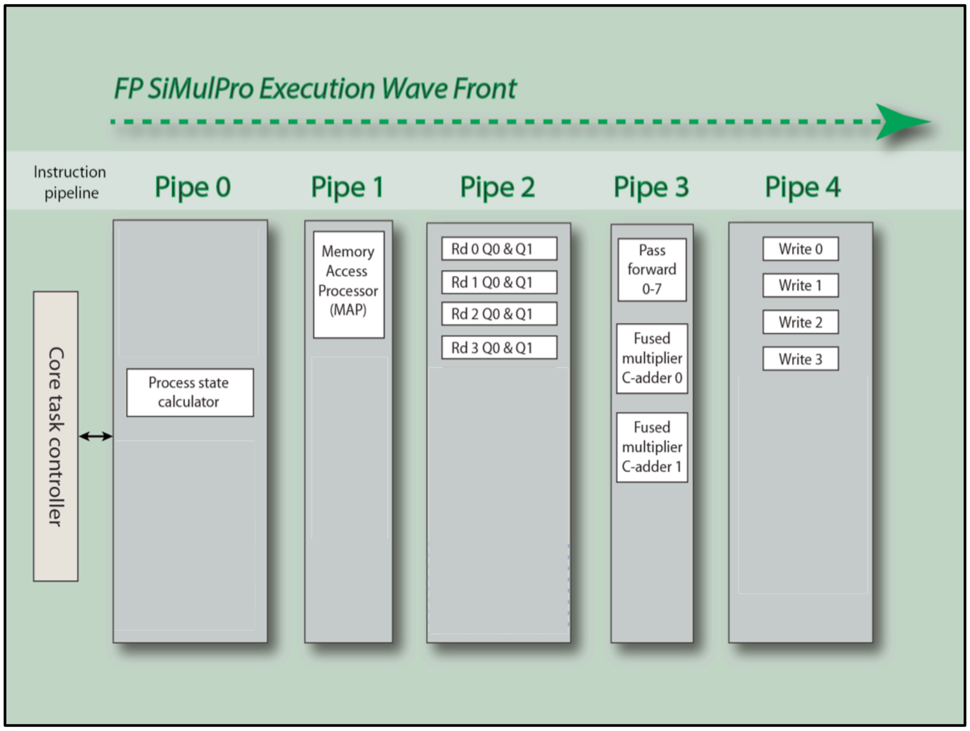
Semantic compatibility tends to require, a fused multiply-accumulate capability, in Floating Point operations.
Fused Multiplier C-Adders, support not only Floating Point, but also Posit Arithmetic, in at least three precisions 64, dual 32, & quad 16 bit.
Algorithms can go from coarse (16 bit), to finer and finest arithmetic (64 bit posit) with reduced communication overhead and memory access for early iterations.
Neural networks can operate in 16 bit mode, providing 8 Floating Posit multiply-accumulates per execution wave front.


FP SiMulPro Execution Wave Front
Instruction pipeline
Pipe 0
Pipe 1
Pipe 2
Pipe 3
Pipe 4
Core task controller
Memory Access Processor (MAP)
Fused multiplier C-adder 0
Fused multiplier C-adder 1

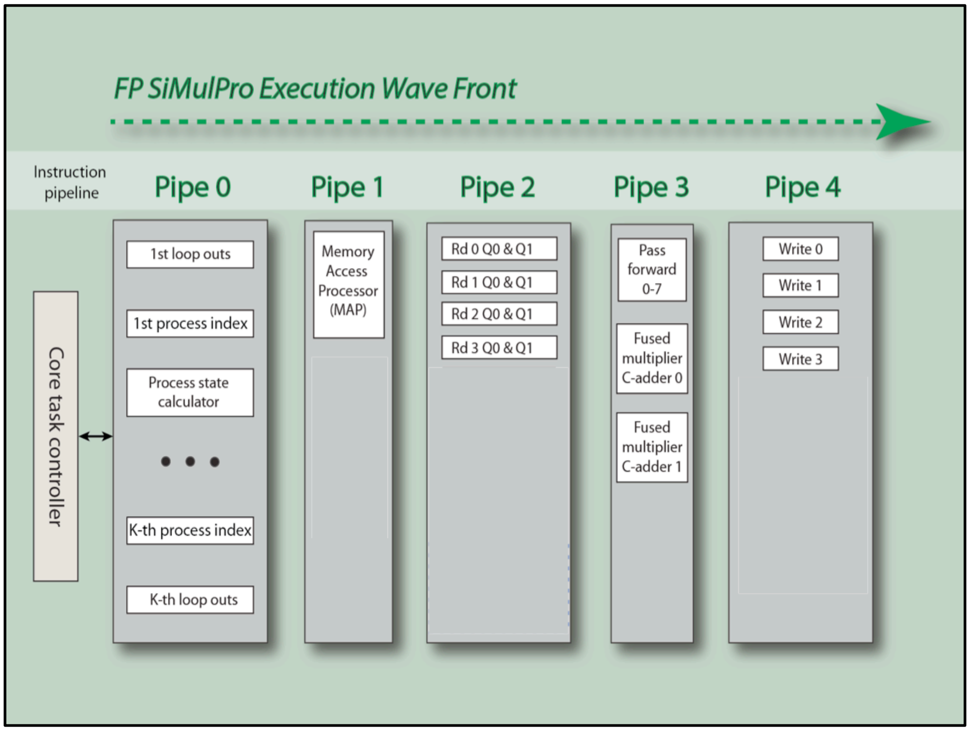
The Memory Access Processor supports all basic, access operations.
So during BLAS 3 local operations, as in Linpack, **all NON-floating point cores** can be turned off



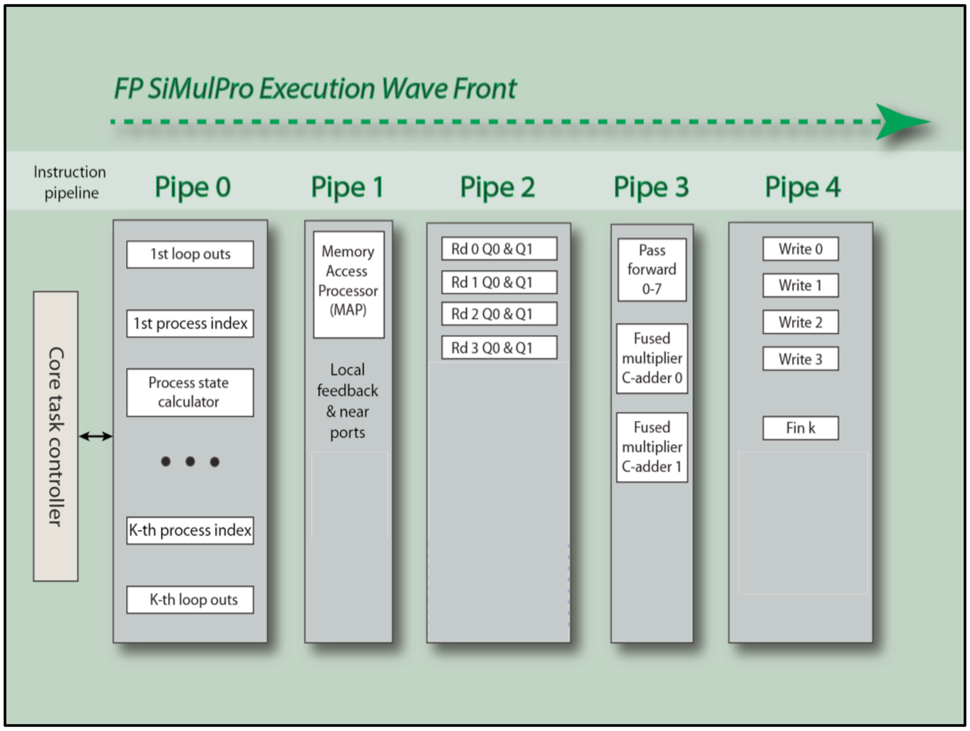
The FP Ram is organized as 8 static ram blocks, of 512 words, each with one read, and one write port.
These RAM can be programmed, to limit access collisions for local sparse matrix operations, to about 2% of the time.
Each of the read ports, supports two queues, so that these collisions, do not significantly stall the multipliers.



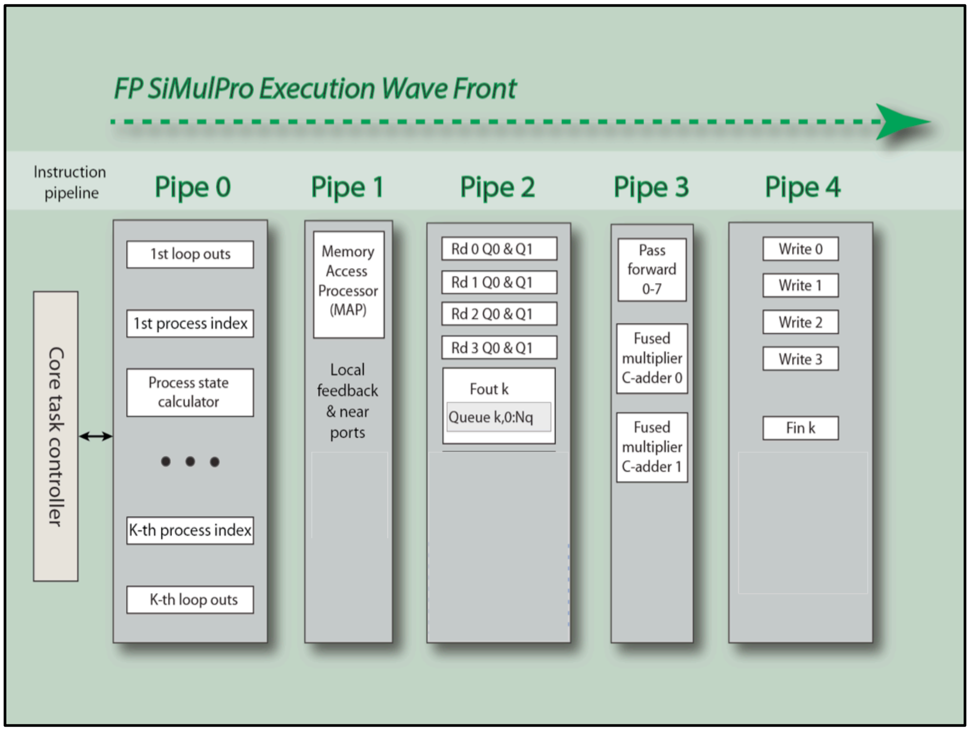
On each clock cycle, the process state calculator generates



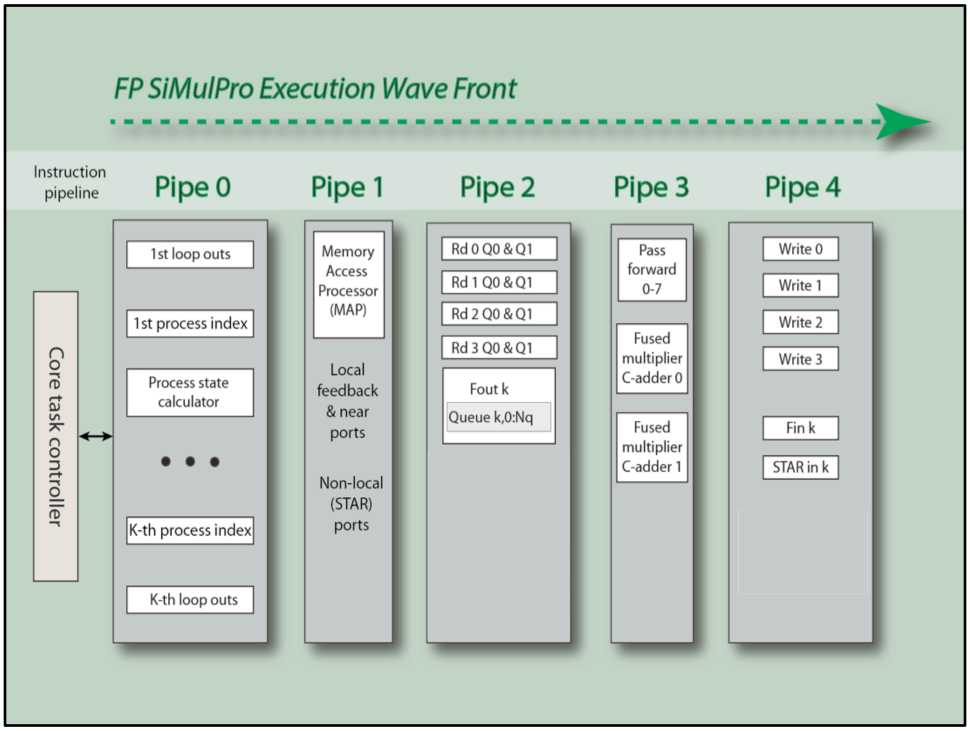
multiple process states (or indexes) and their corresponding loop outputs.
Assume each instructed resource includes 256 local instructions per task.
If k is 4, this is a virtual, VLIW instruction space of 4 Giga instructions.
If k is 6, this is 256 Tera instructions.



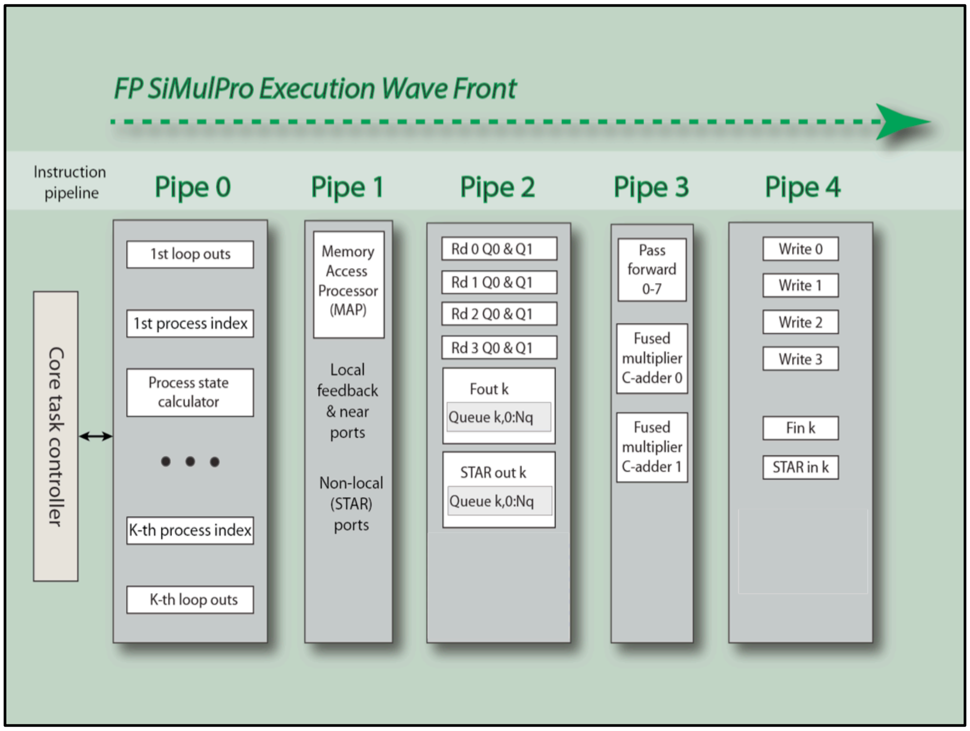
Data is fed back, from instruction pipe 4



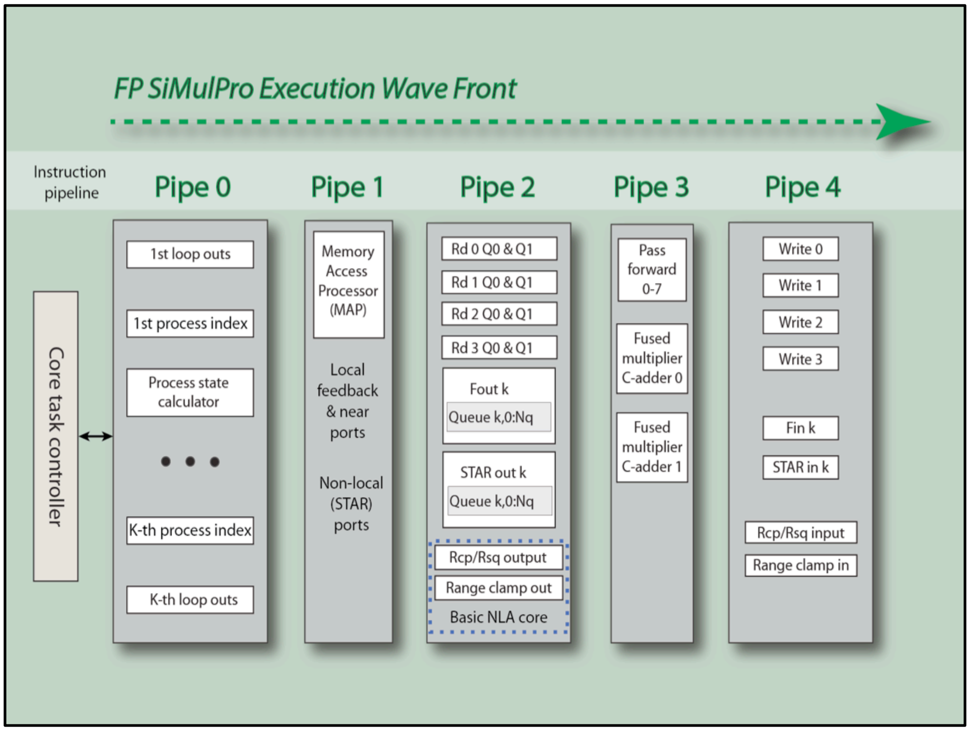
to output queues in pipe 2.
This provides a router-less **feed** interface to neighboring modules, less hardware = less energy use.



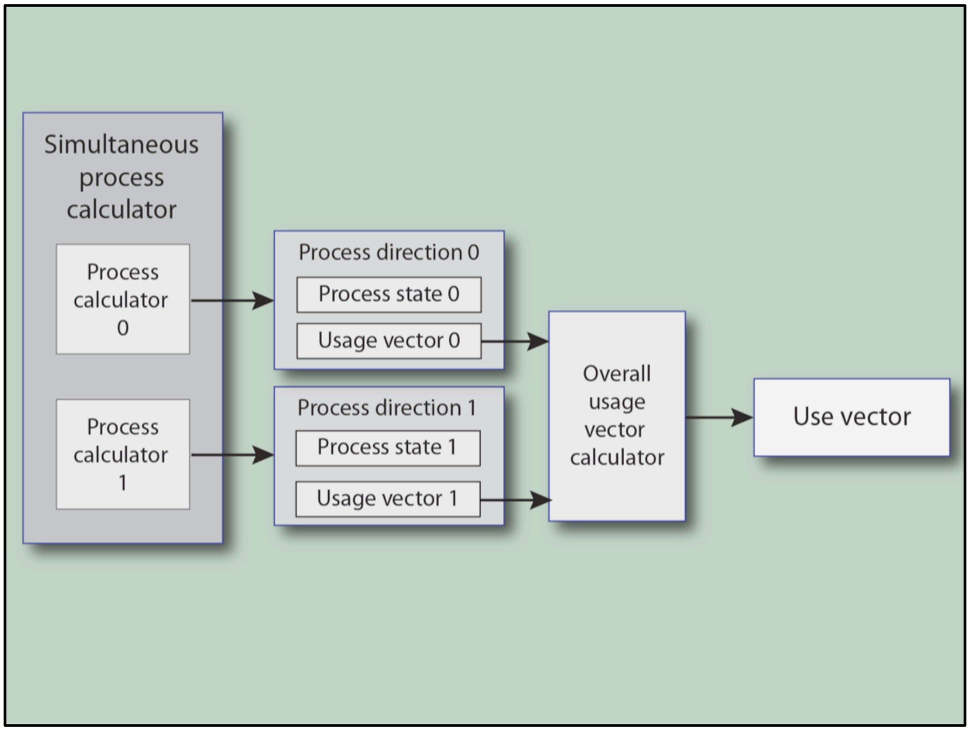
Farther communications uses the STAR input, which is my second presentation,



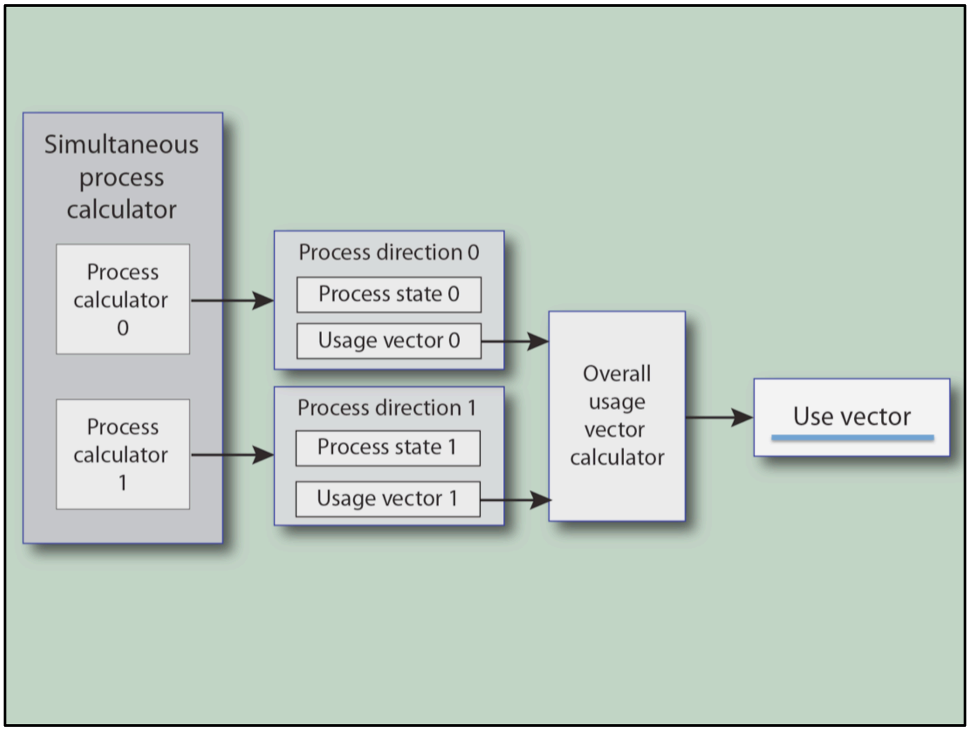
and output data ports.
Process state changes, are based upon data availability,
  from local sources,
  or from elsewhere in a chip,
  or across a computer floor.



Reciprocal and Reciprocal square roots are supported.
A range clamp circuit supports range limiting for computing transcendental functions.
Integer to float and float to integer are also supported.



Within each execution wave front,



Simultaneous process calculator

Process calculator 0

Process calculator 1

Process direction 0

Process state 0

Usage vector 0

Process direction 1

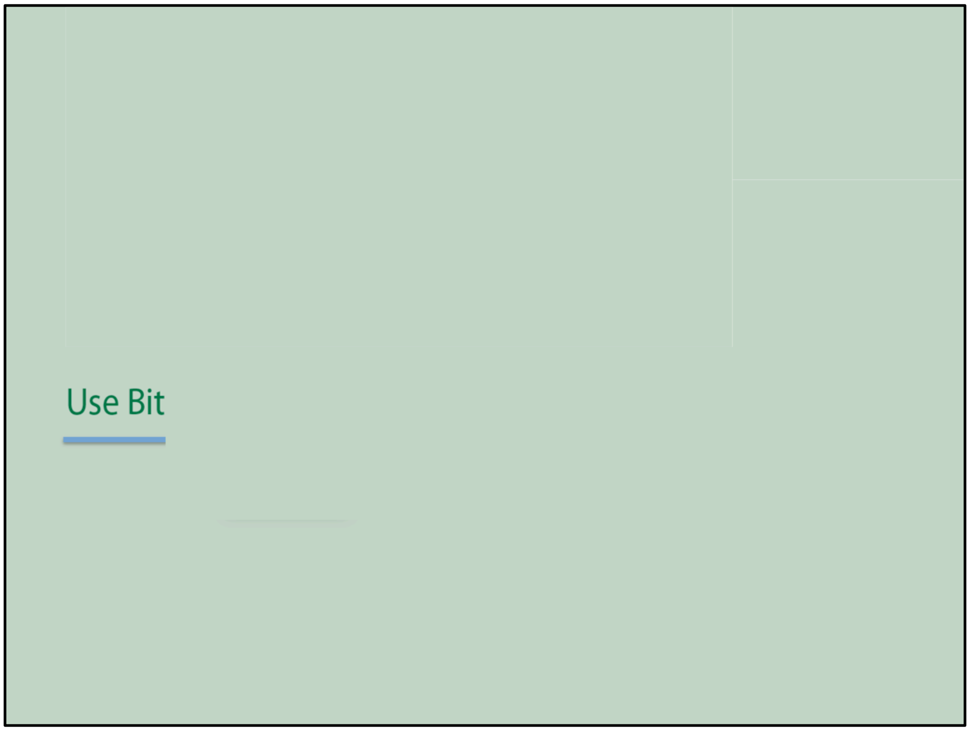
Process state 1

Usage vector 1

Overall usage vector calculator

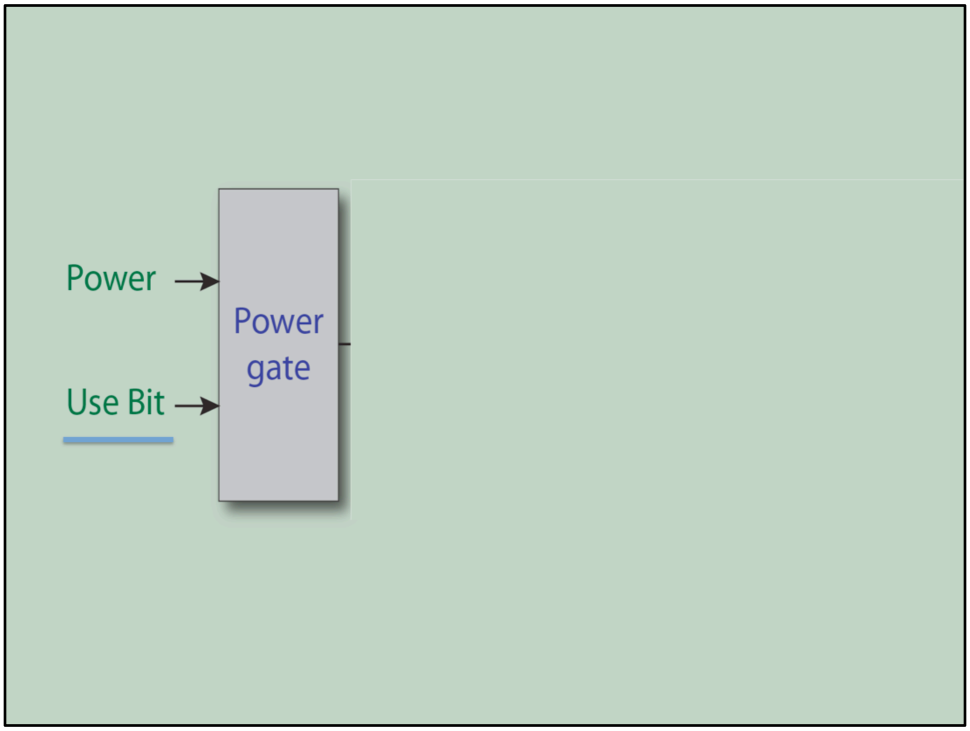
Use vector

a use vector is generated, of the used resources.

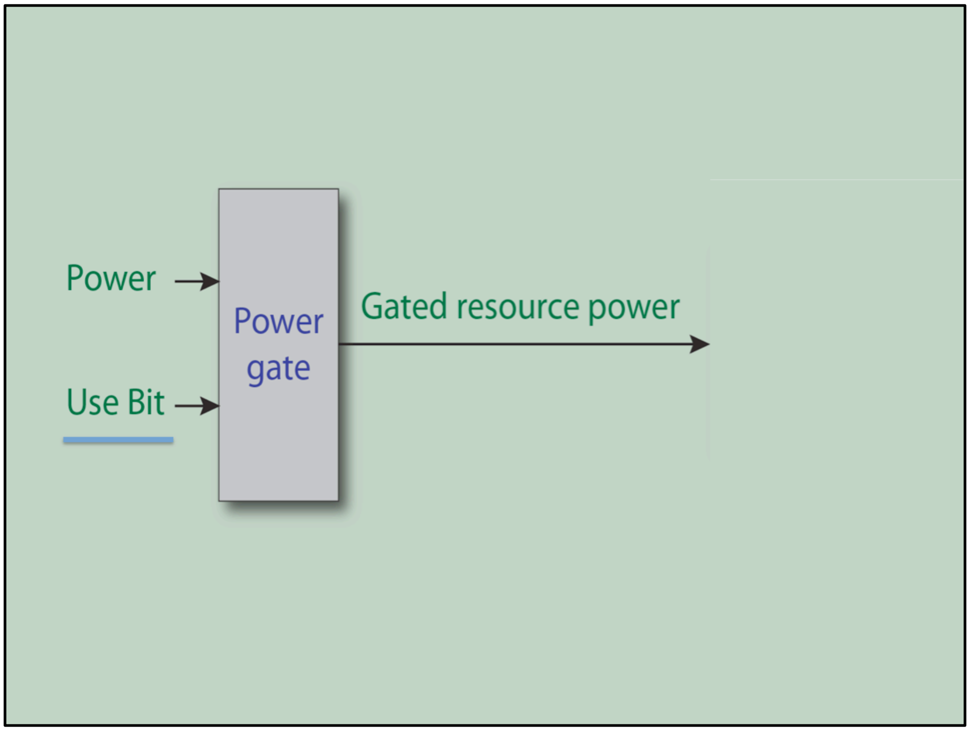


Use Bit

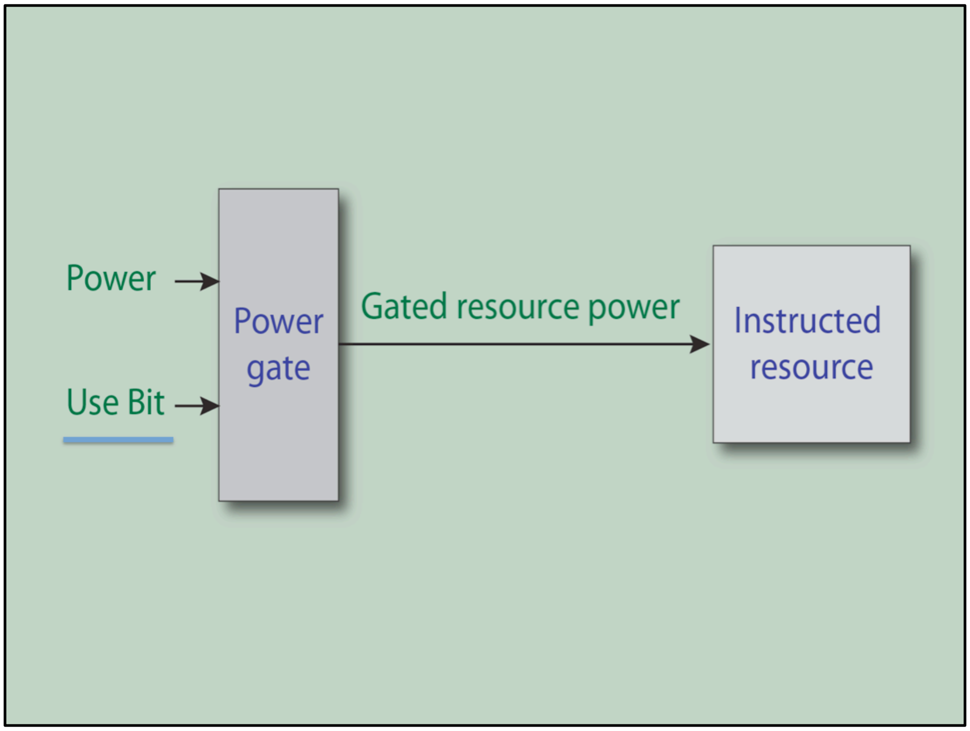
Within that wave front, a use bit



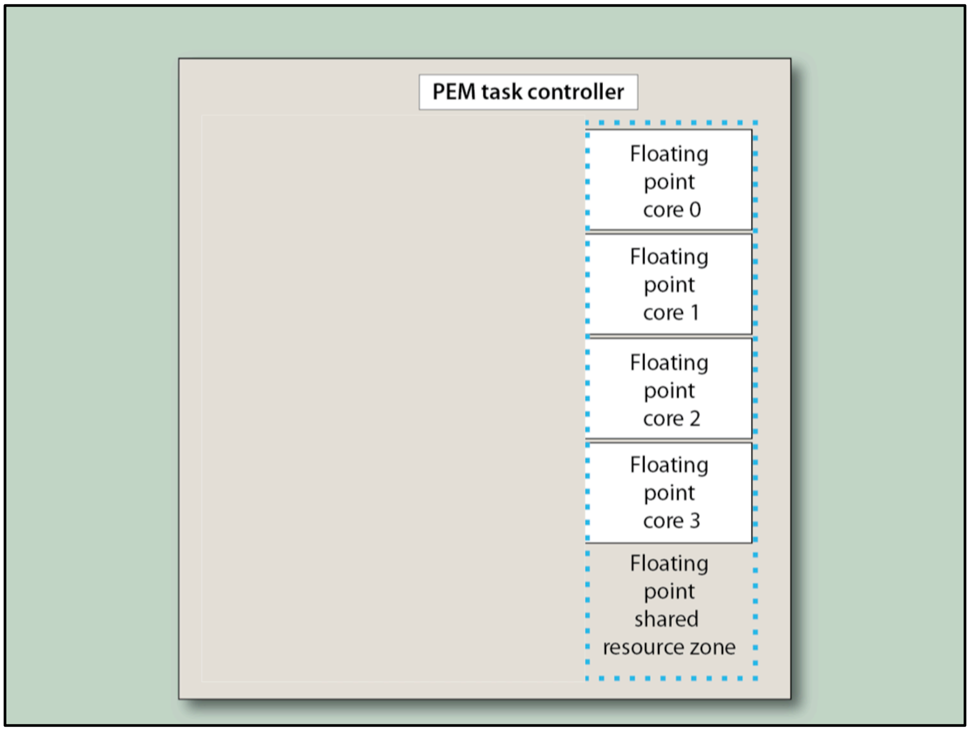
drives a power gate



generating a gated power



for each instructed resource.
As the execution wave front traverses the resource, the gated power is used by that resource.
In CMOS, the clock may be gated to control power.



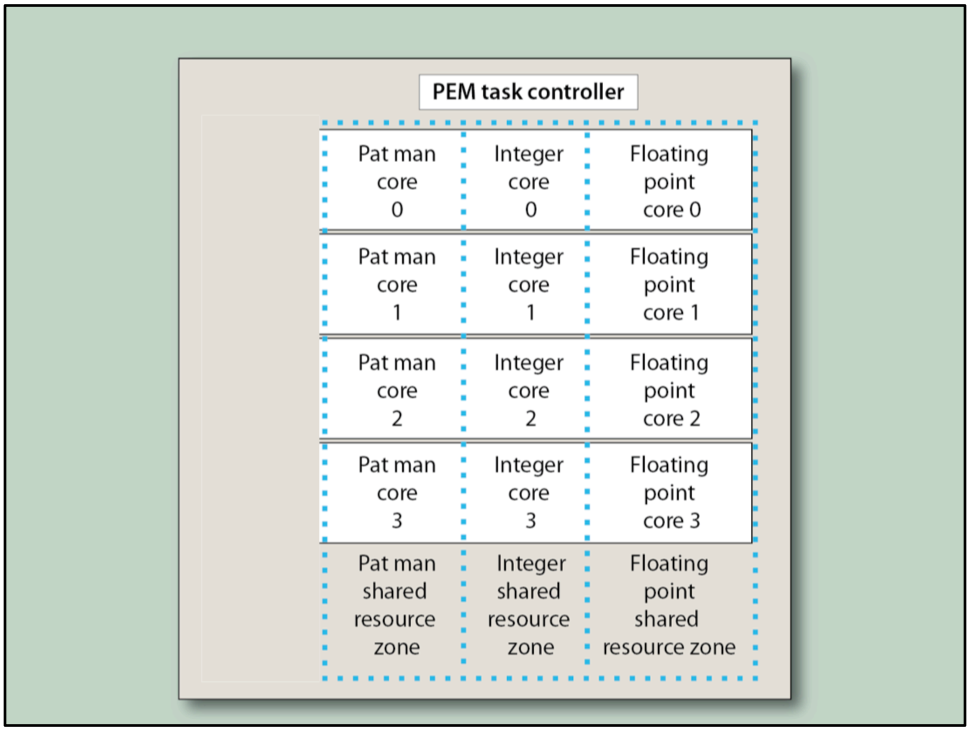
PEM stands for Programmable Execution Module, roughly equivalent to a quad core superscalar microprocessor,

without the overhead.

Instances of the same typed cores, such as Floating Point cores, can share their instructed resources. This increases the virtual VLIW space.

Assume that 4 cores, support 6 simultaneous processes, and 256 instructions per instructed resource.

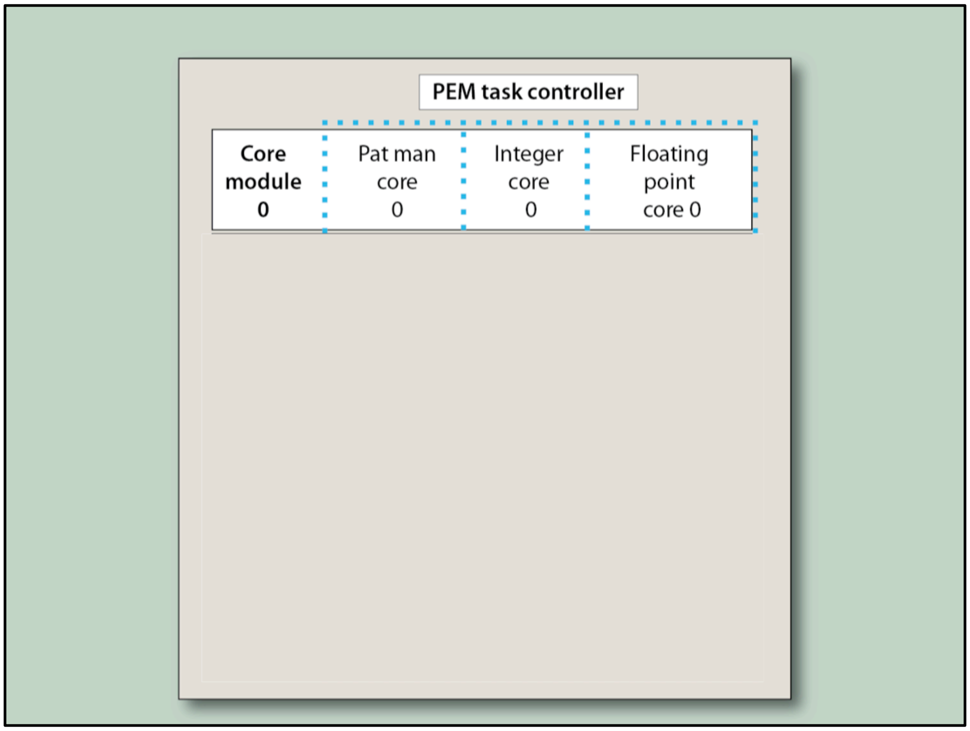
The VLIW instruction space is 2^(24*8) = 2^ 192 = 4 E57



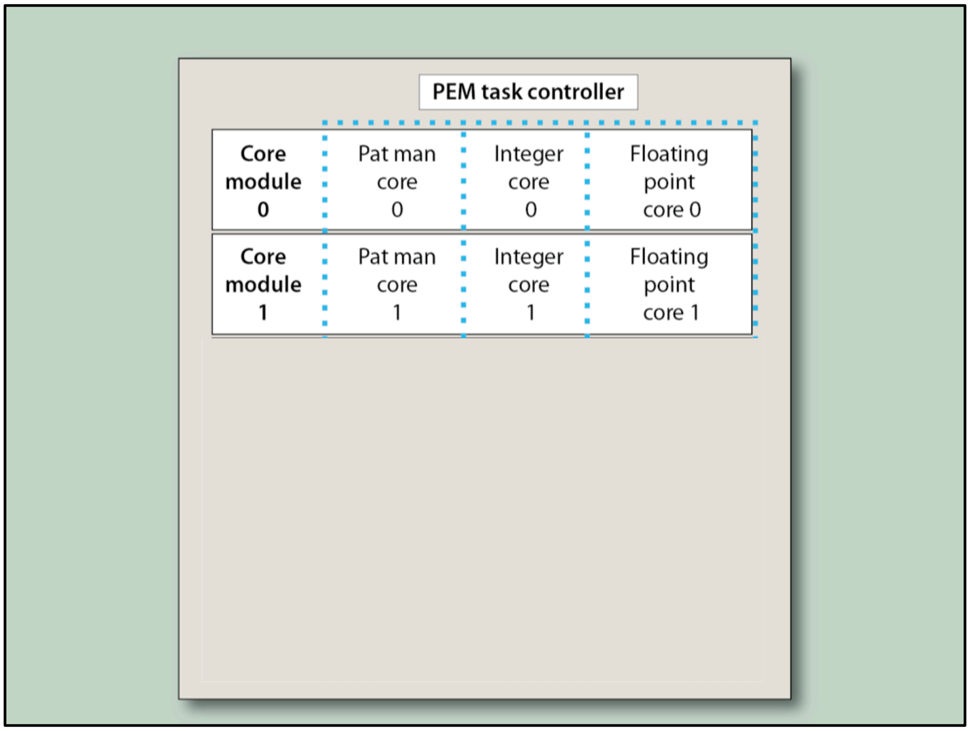
PEM task controller

Pat man core 0	Integer core 0	Floating point core 0
Pat man core 1	Integer core 1	Floating point core 1
Pat man core 2	Integer core 2	Floating point core 2
Pat man core 3	Integer core 3	Floating point core 3
Pat man shared resource zone	Integer shared resource zone	Floating point shared resource zone

There are 3 types of cores in each PEM, floating point, integer and a pattern manager (Pat man) core.
The integer core implements the necessary arithmetic required for application compatibility,
and also keeps track of the indexing associated with every word of the floating point memory,
for local sparse matrix operations.
Pat man organizes and directs Floating Point and integer activities needed for these operations.



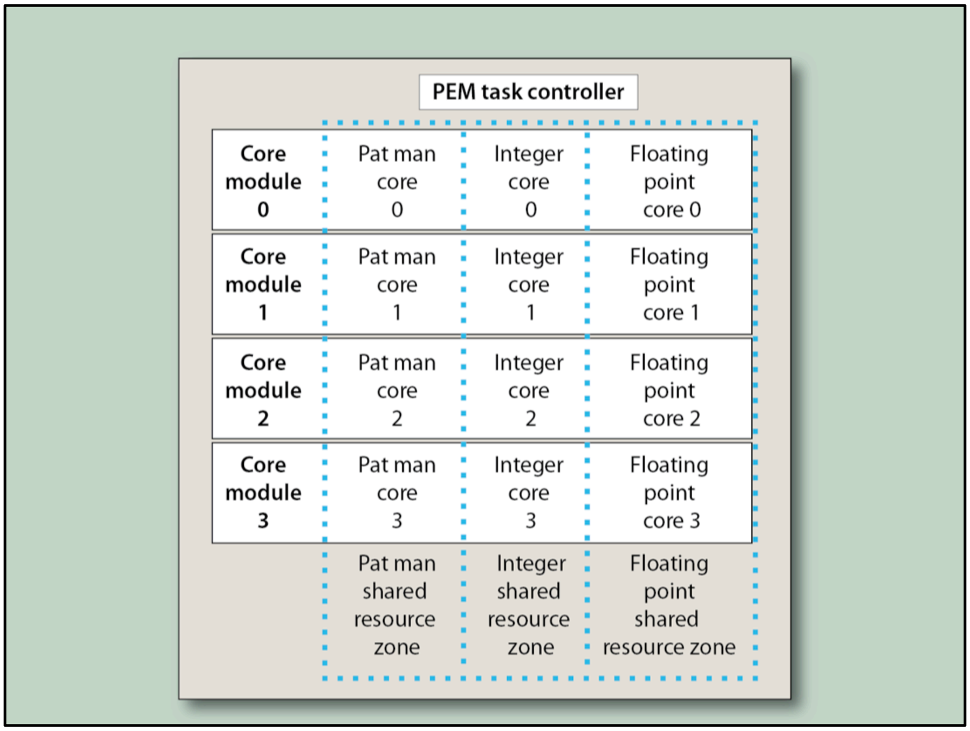
Each core module, with its three cores, can operate as a unit.
Core module 0 may be performing a simulation of one object.



PEM task controller

Core module 0	Pat man core 0	Integer core 0	Floating point core 0
Core module 1	Pat man core 1	Integer core 1	Floating point core 1

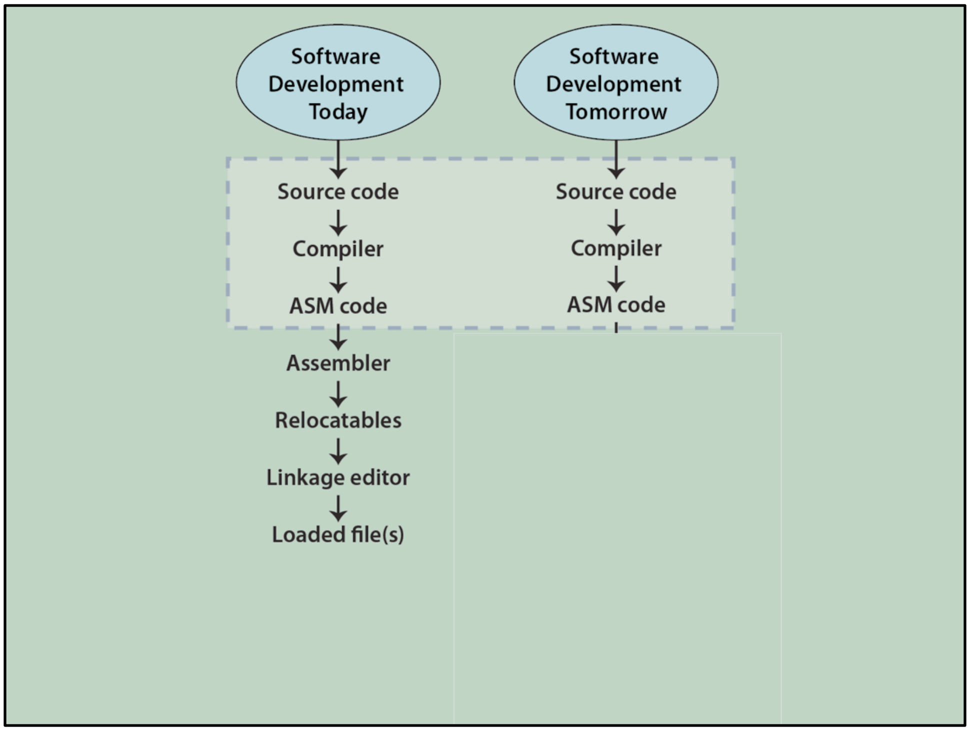
While core module 1 may be simulating a second kind of model.



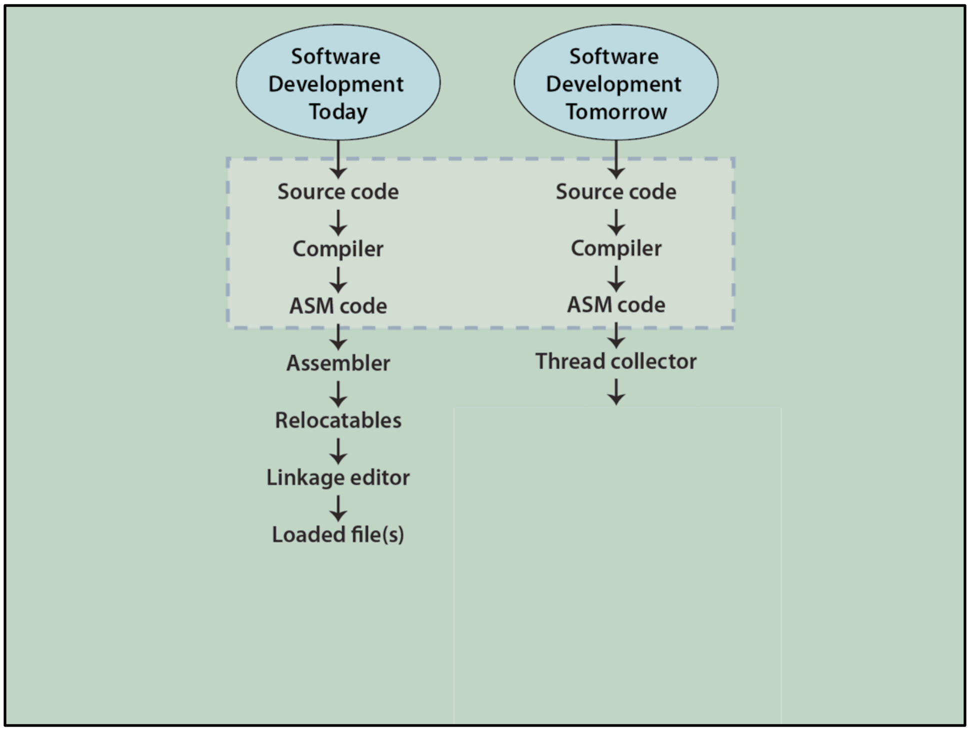
PEM task controller

Core module 0	Pat man core 0	Integer core 0	Floating point core 0
Core module 1	Pat man core 1	Integer core 1	Floating point core 1
Core module 2	Pat man core 2	Integer core 2	Floating point core 2
Core module 3	Pat man core 3	Integer core 3	Floating point core 3
	Pat man shared resource zone	Integer shared resource zone	Floating point shared resource zone

Alternatively, each core module may be configured to model the same object, but at differing geometric locations.
Let's look briefly at software development.



The compiler is basically unchanged between software development, today and tomorrow.



Software Development Today

Source code

Compiler

ASM code

Assembler

Relocatables

Linkage editor

Loaded file(s)

Software Development Tomorrow

Source code

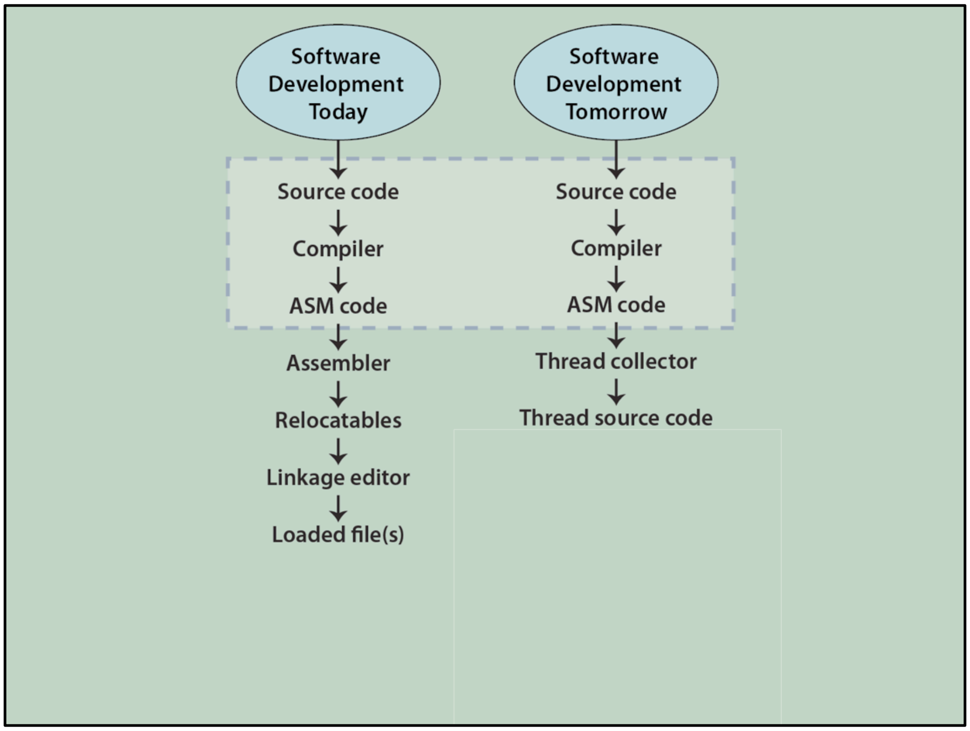
Compiler

ASM code

Thread collector

The superscalar interpreter hardware, is transformed into the thread collector, which translates assembler instructions into micro-code.

The microcode is then scheduled, as close to the start of the thread, as allowed by its preceding, assembler instructions.



Software Development Today

Source code

Compiler

ASM code

Assembler

Relocatables

Linkage editor

Loaded file(s)

Software Development Tomorrow

Source code

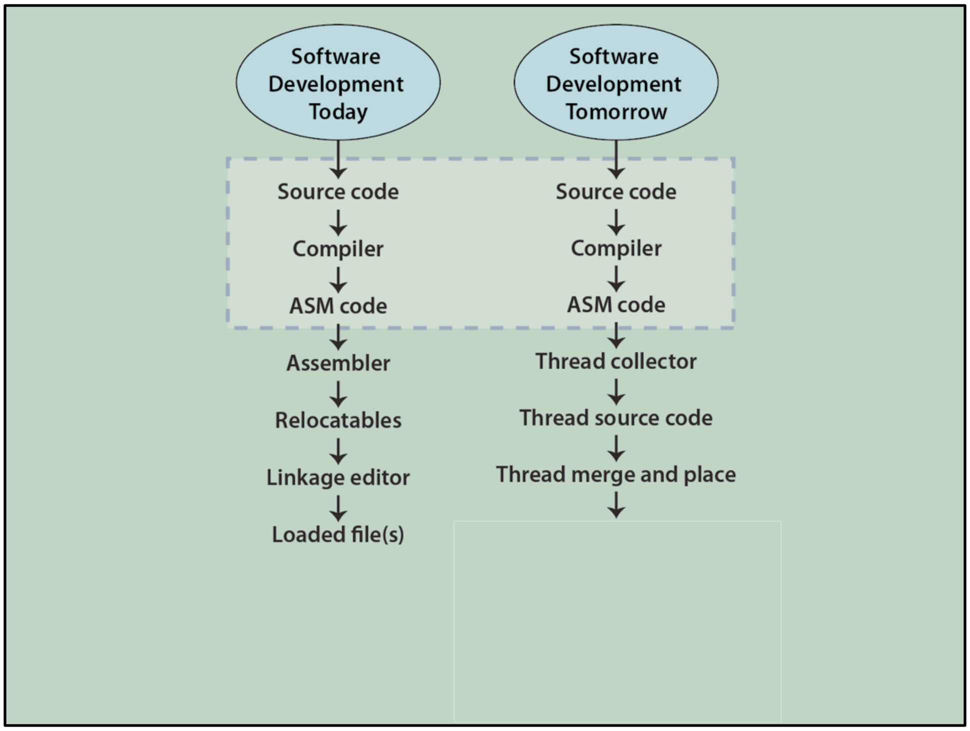
Compiler

ASM code

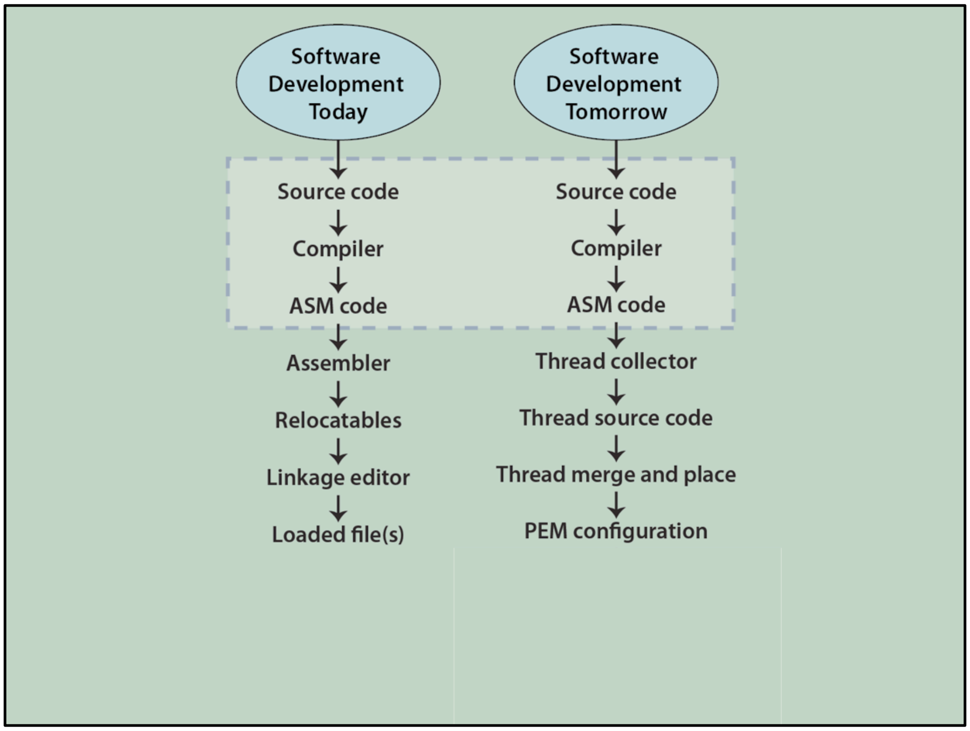
Thread collector

Thread source code

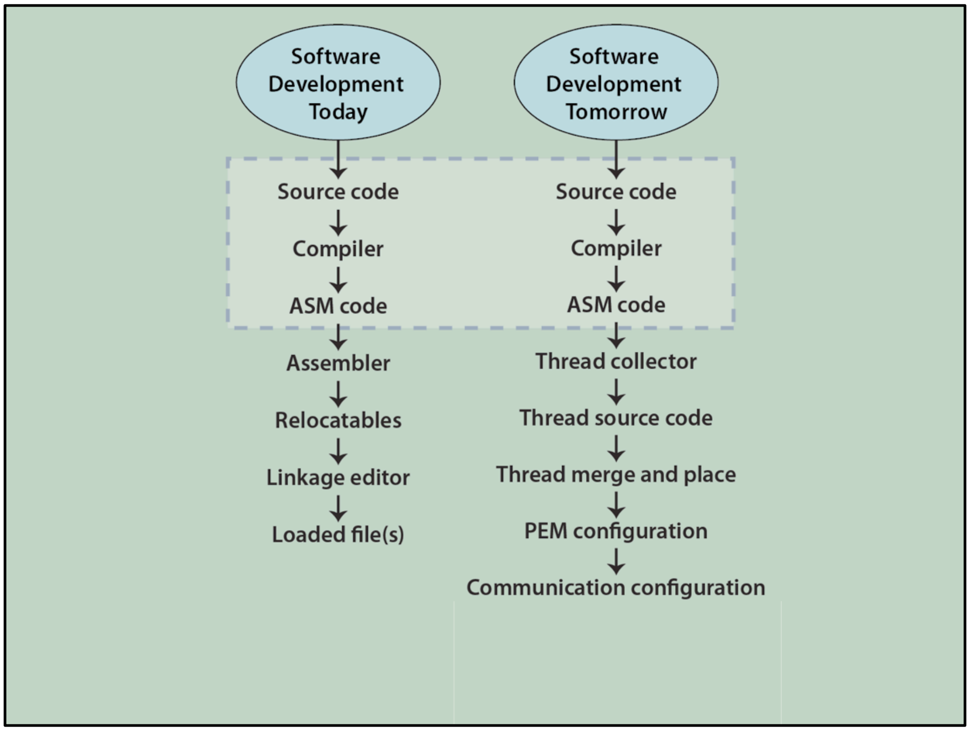
This utility outputs one or more thread, source code, files.



The multi-thread controller, becomes a software utility, which merges and places the threads



into configurations of the PEM.



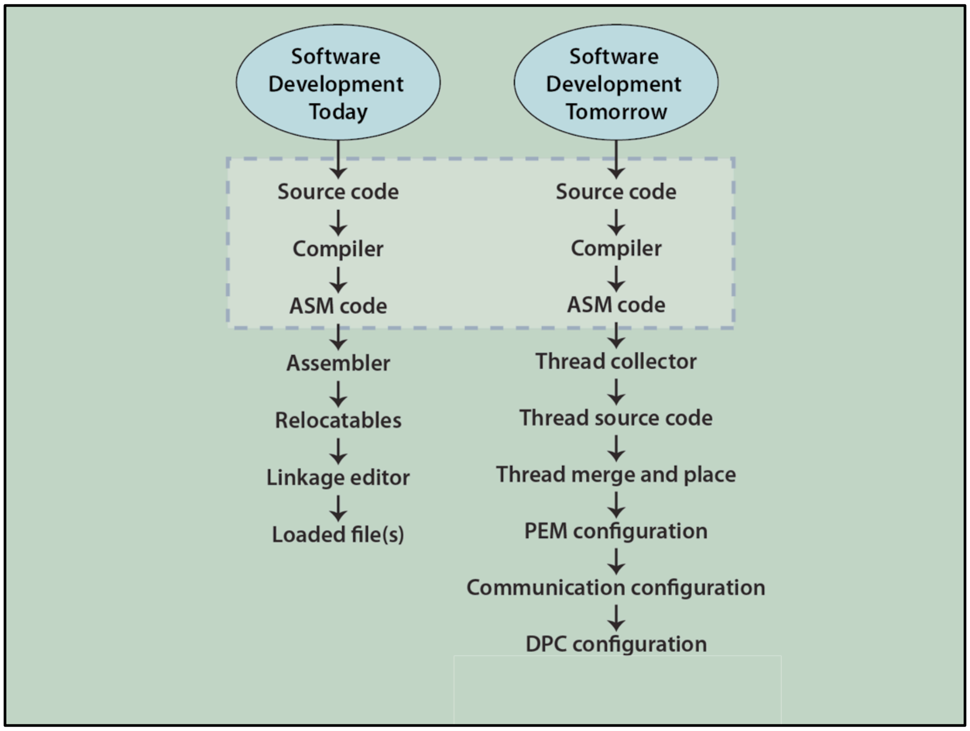
Software Development Today

Source code → Compiler → ASM code → Assembler → Relocatables → Linkage editor → Loaded file(s)

Software Development Tomorrow

Source code → Compiler → ASM code → Thread collector → Thread source code → Thread merge and place → PEM configuration → Communication configuration

The farther network, and router-less, neighbor communications, are then configured



Software Development Today

Source code

Compiler

ASM code

Assembler

Relocatables

Linkage editor

Loaded file(s)

Software Development Tomorrow

Source code

Compiler

ASM code

Thread collector

Thread source code

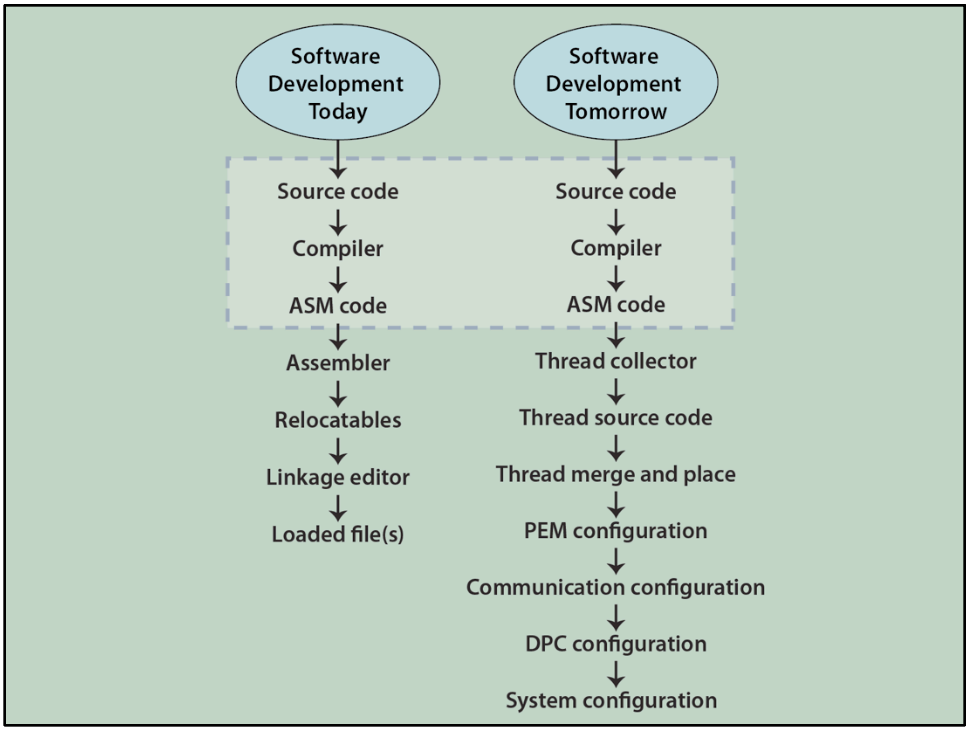
Thread merge and place

PEM configuration

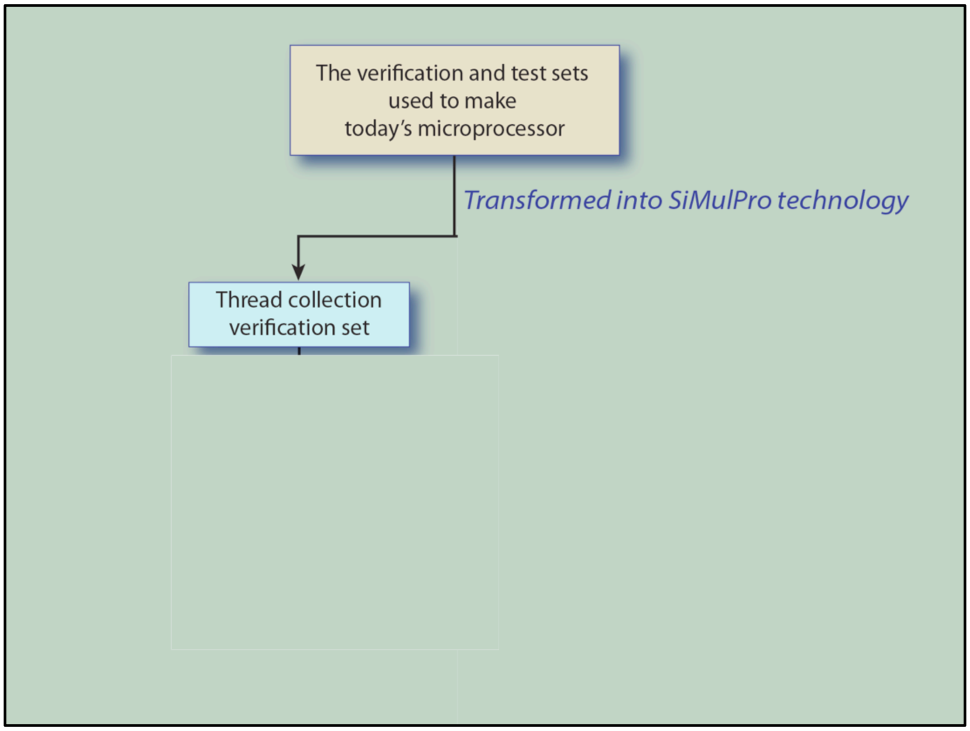
Communication configuration

DPC configuration

for each Data Processor Chip in the system.



Configuring the system also involves
Configuring anticipating, memory controllers, interfaced to DRAM.
Configuring communication within the Data Processor Chips to across the system,
As well as communicating with the Data Center, in which the supercomputer resides.

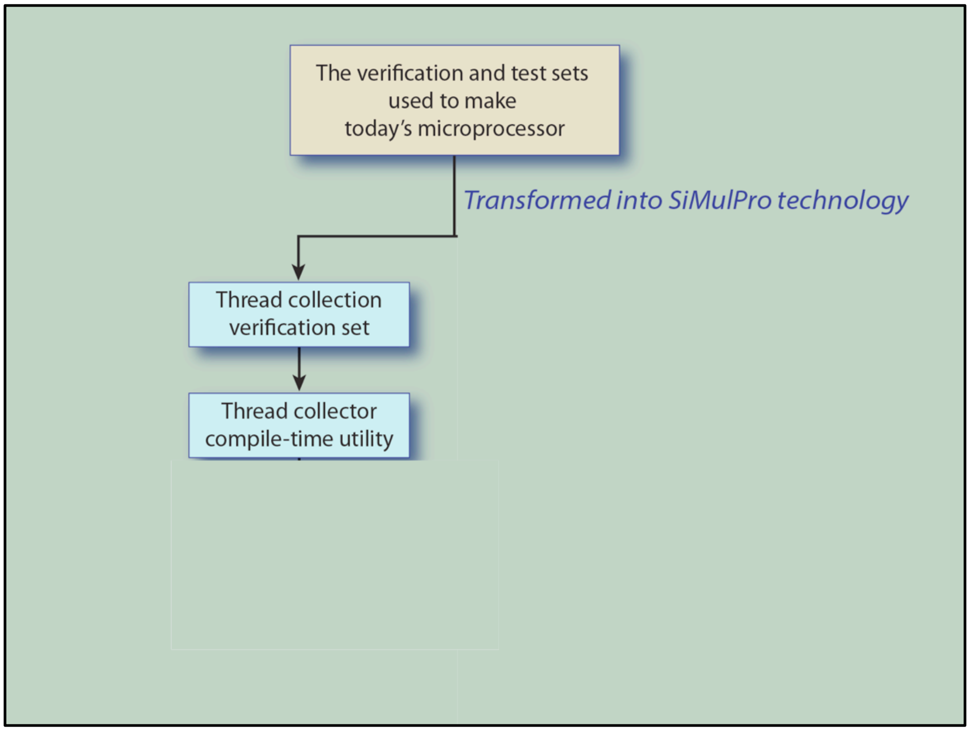


The verification and test sets used to make today's microprocessor

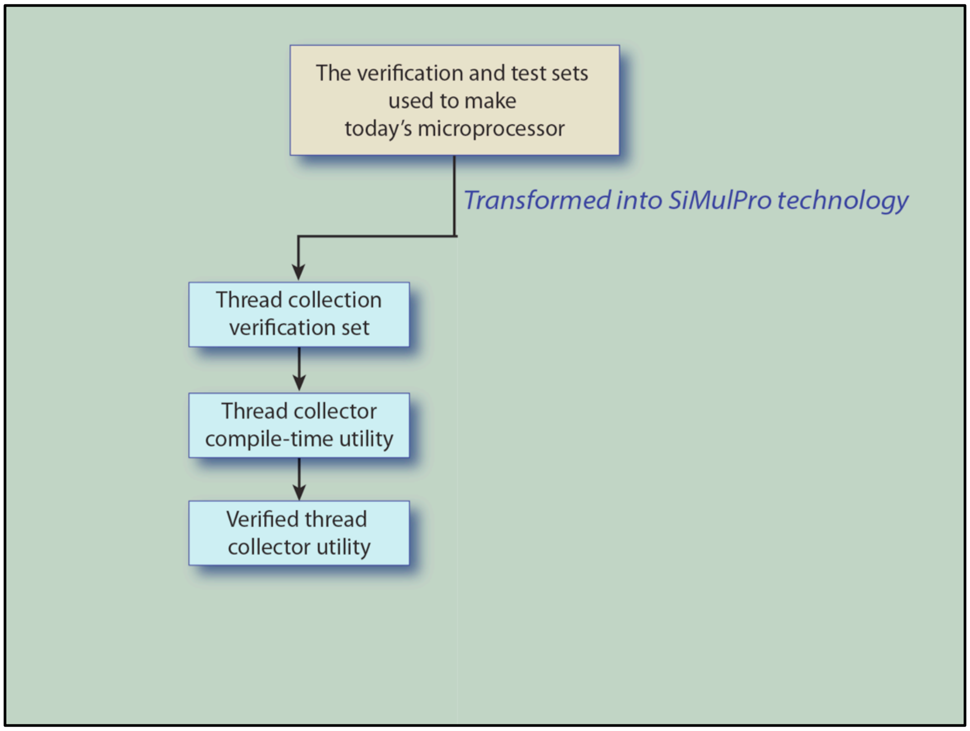
Transformed into SiMulPro technology

Thread collection verification set

The verification and test sets for today's, superscalar instruction interpreters, become the thread collection, verification set.


The verification and test sets used to make today's microprocessor
Transformed into SiMulPro technology
Thread collection verification set
Thread collector compile-time utility

The superscalar interpreter's behavioral model is transformed into the thread collector.
This removes the interpreter from hardware, and **its energy budget.**



The verification and test sets used to make today's microprocessor

Transformed into SiMulPro technology

Thread collection verification set

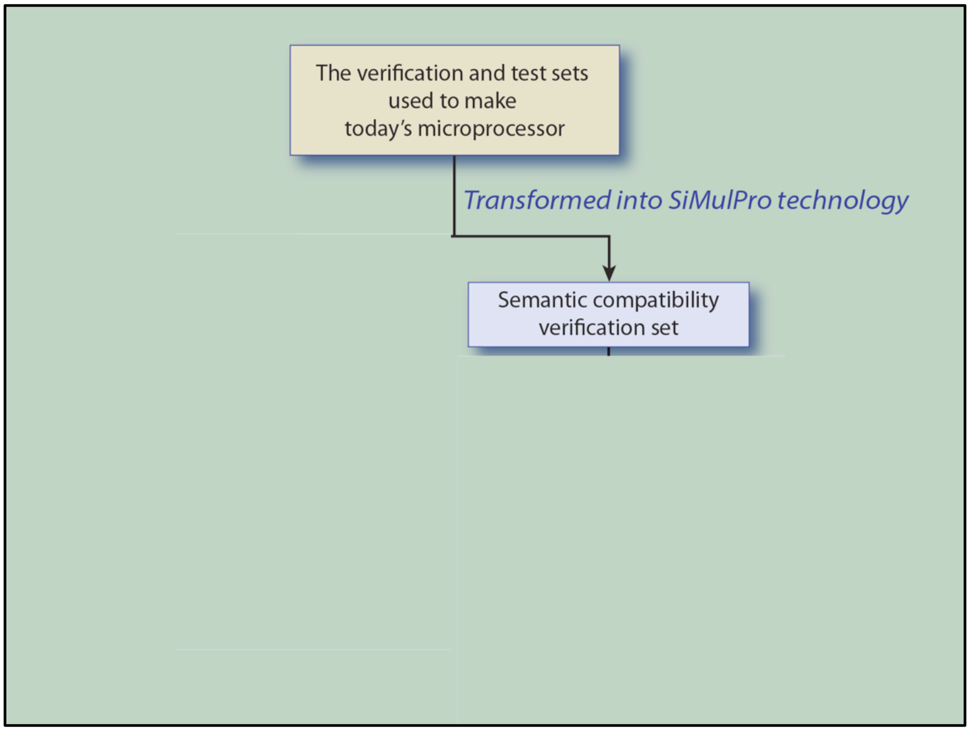
Thread collector compile-time utility

Verified thread collector utility

Exercising this verification set,
and the utility,
generates a verified thread collector.

The verification and test sets used to make today's microprocessor

The microprocessor is thoroughly exercised by its verification and test sets.

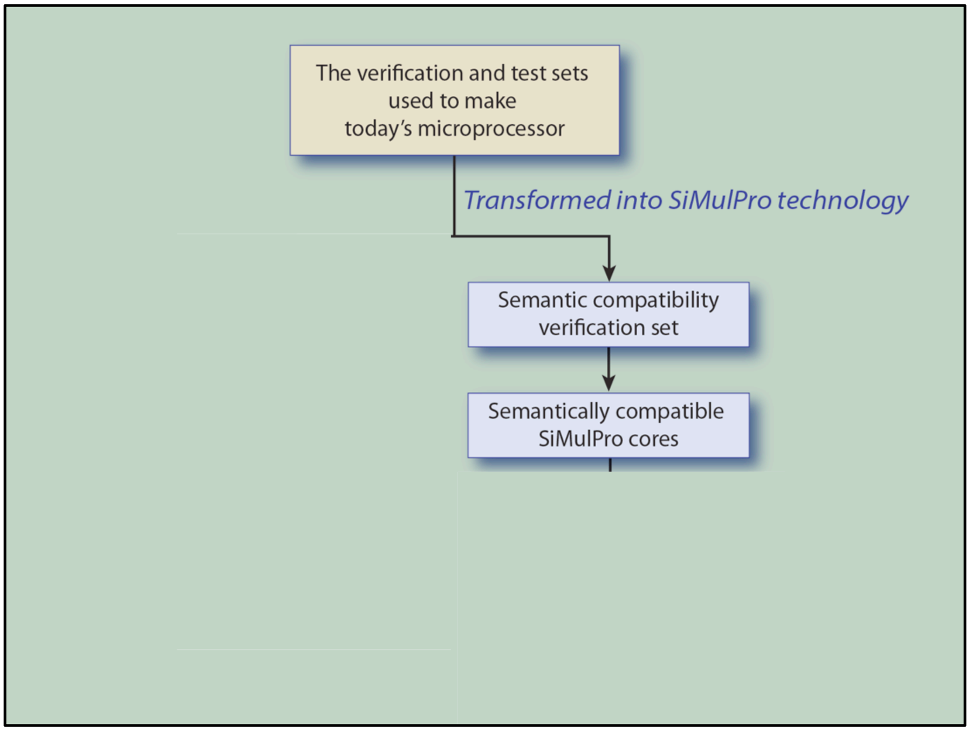


The verification and test sets used to make today's microprocessor

Transformed into SiMulPro technology

Semantic compatibility verification set

This is transformed into the semantic compatibility, verification set for the new, software development tools.



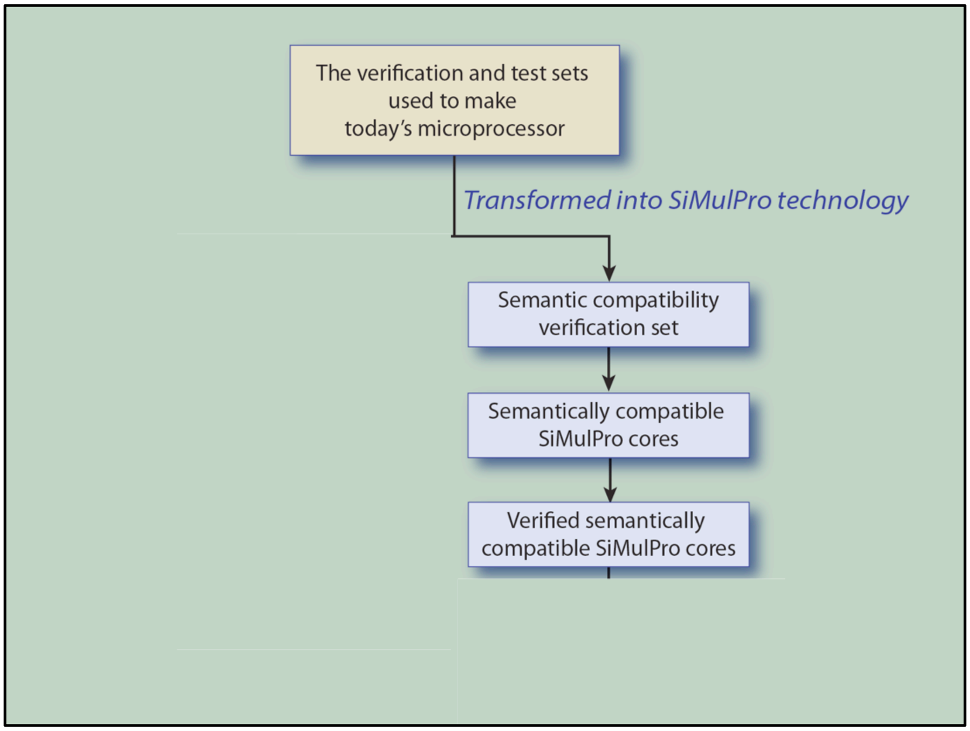
The verification and test sets used to make today's microprocessor

Transformed into SiMulPro technology

Semantic compatibility verification set

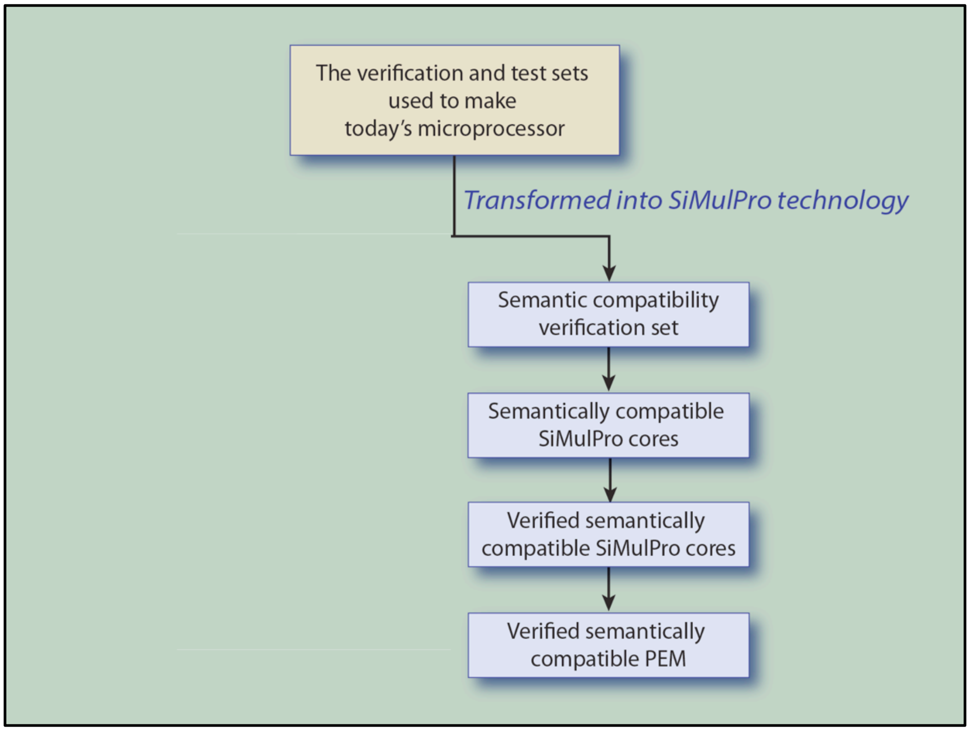
Semantically compatible SiMulPro cores

Behavioral models of the data processing resources are injected into templates, to create the core module.

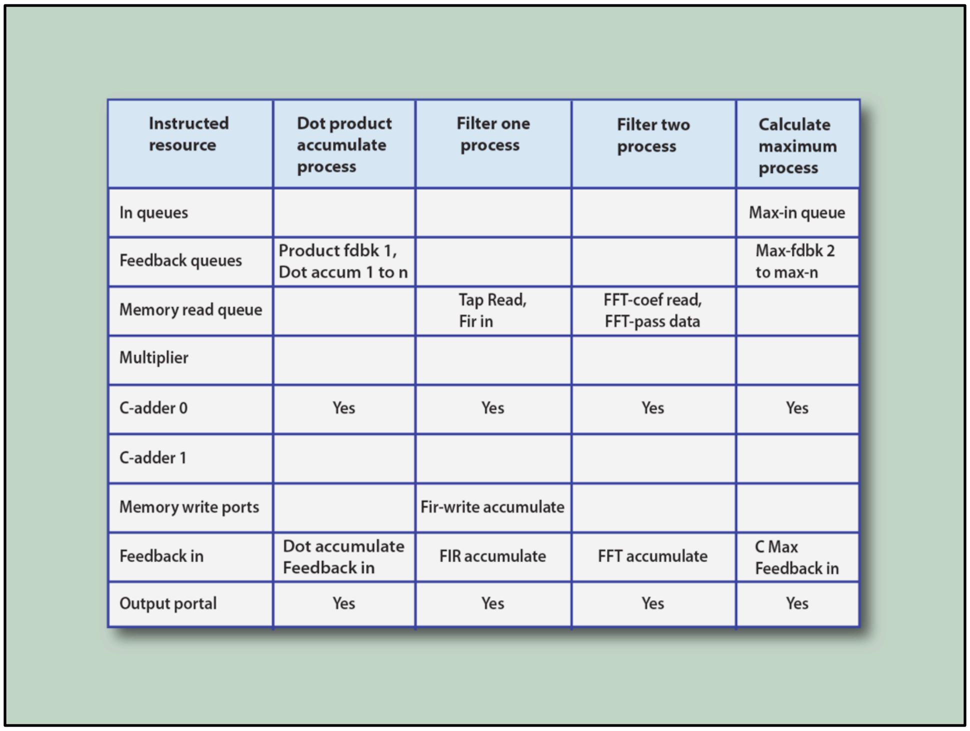


The verification and test sets used to make today's microprocessor
Transformed into SiMulPro technology
Semantic compatibility verification set
Semantically compatible SiMulPro cores
Verified semantically compatible SiMulPro cores

Exercising the verification set with the core module generates the verified, semantically compatible core module.

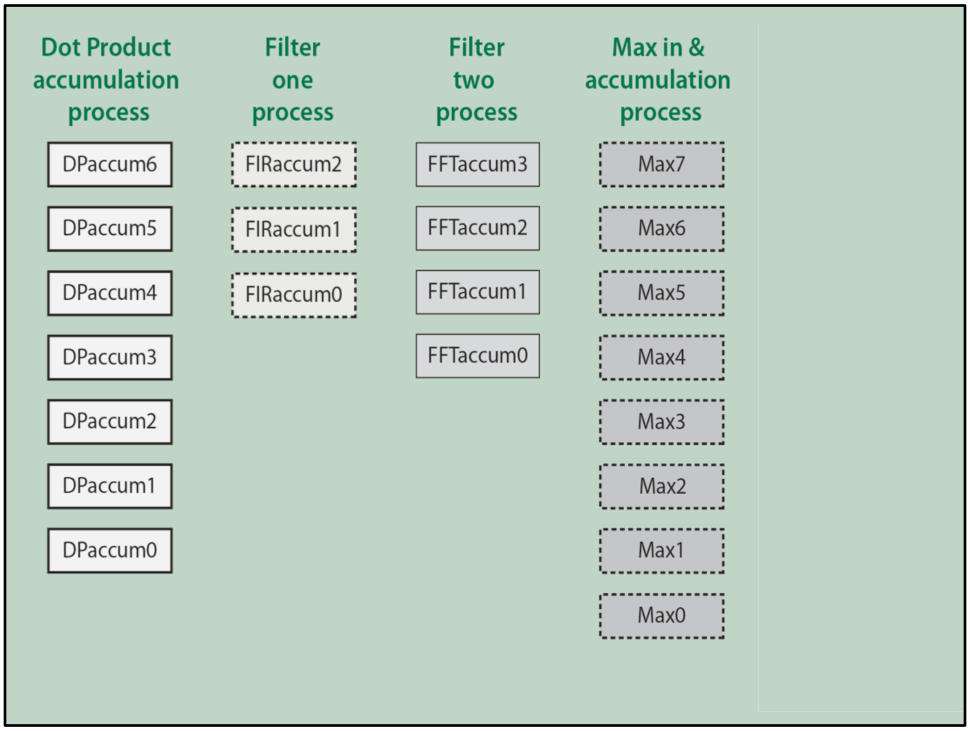

The verification and test sets used to make today's microprocessor
Transformed into SiMulPro technology
Semantic compatibility verification set
Semantically compatible SiMulPro cores
Verified semantically compatible SiMulPro cores
Verified semantically compatible PEM

Verifying these core modules creates the semantically compatible PEM.
Today, a thread of execution, is the smallest sequence of program instructions, which can be managed independently, by a scheduler, in an operating system.



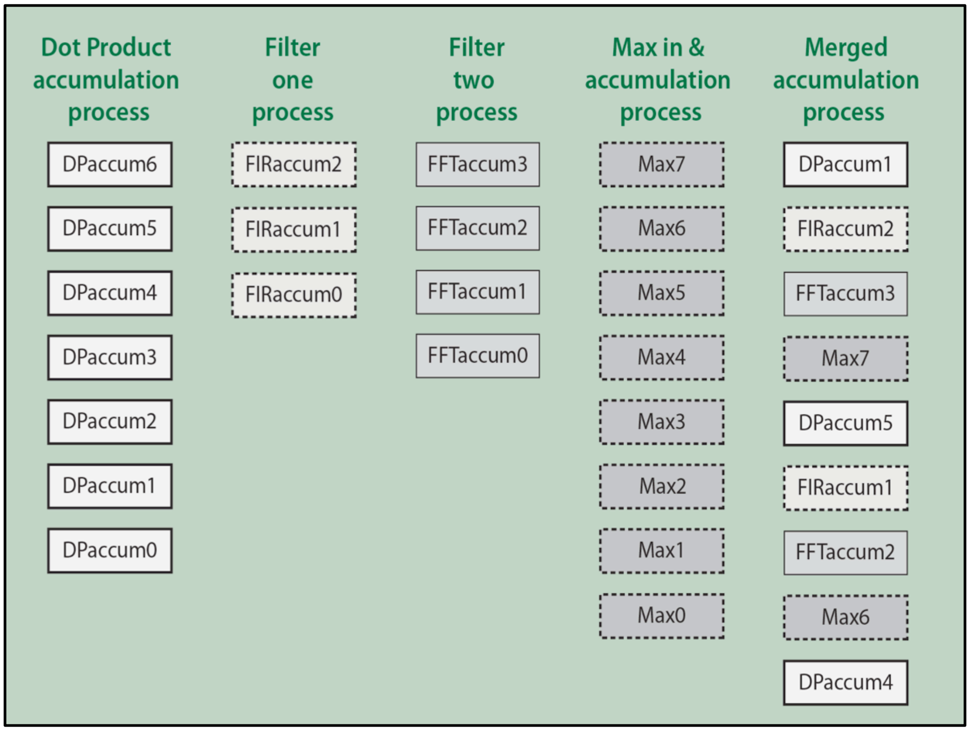
Instructed resource	Dot product accumulate process	Filter one process	Filter two process	Calculate maximum process
In queues				Max-in queue
Feedback queues	Product fdbk 1, Dot accum 1 to n			Max-fdbk 2 to max-n
Memory read queue		Tap Read, Fir in	FFT-coef read, FFT-pass data	
Multiplier				
C-adder 0	Yes	Yes	Yes	Yes
C-adder 1				
Memory write ports		Fir-write accumulate		
Feedback in	Dot accumulate Feedback in	FIR accumulate	FFT accumulate	C Max Feedback in
Output portal	Yes	Yes	Yes	Yes

In this architecture,
threads are operations of at least one core, in a PEM,
expressed as one or more processes.
A program is ***transformed at compile-time, into the computer's simultaneous processes.***
After collecting the threads,
the intermediate, program representation is analyzed, and thread merging begins.



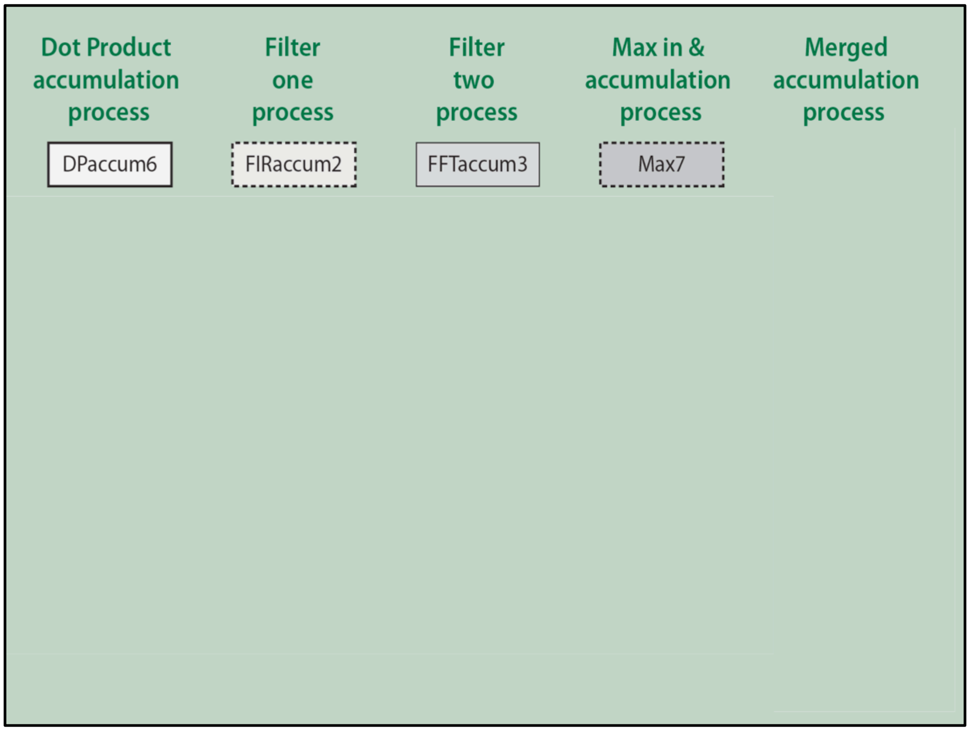
Dot Product accumulation process	Filter one process	Filter two process	Max in & accumulation process
DPaccum6	FIRaccum2	FFTaccum3	Max7
DPaccum5	FIRaccum1	FFTaccum2	Max6
DPaccum4	FIRaccum0	FFTaccum1	Max5
DPaccum3		FFTaccum0	Max4
DPaccum2			Max3
DPaccum1			Max2
DPaccum0			Max1
			Max0

Merging processes is in terms of their respective process states,



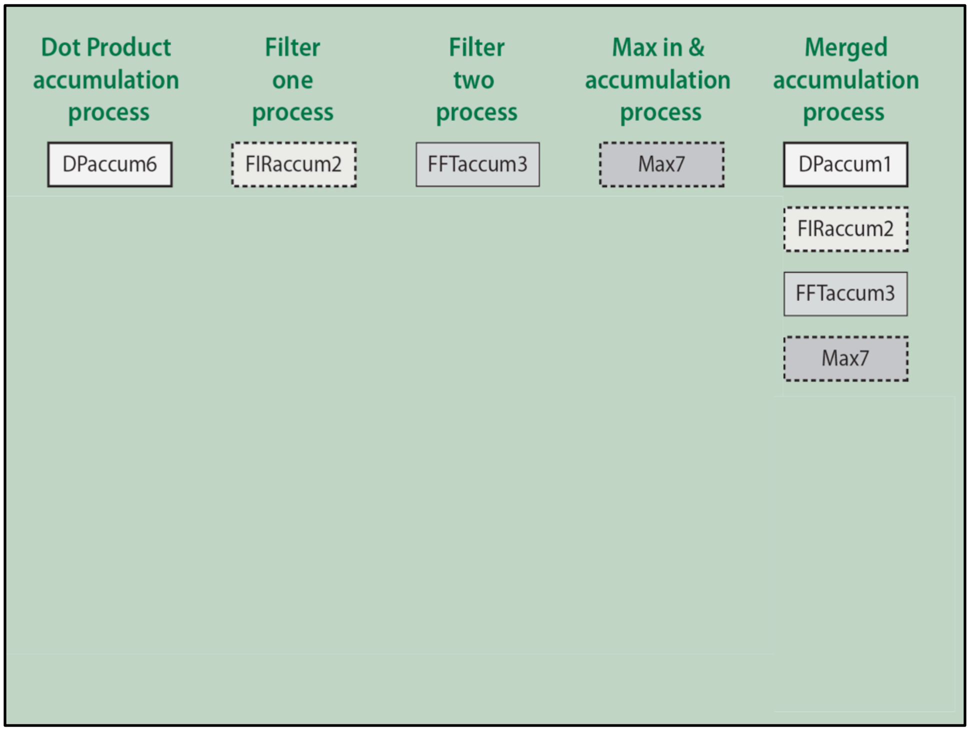
Dot Product accumulation process	Filter one process	Filter two process	Max in & accumulation process	Merged accumulation process
DPaccum6	FIRaccum2	FFTaccum3	Max7	DPaccum1
DPaccum5	FIRaccum1	FFTaccum2	Max6	FIRaccum2
DPaccum4	FIRaccum0	FFTaccum1	Max5	FFTaccum3
DPaccum3		FFTaccum0	Max4	Max7
DPaccum2			Max3	DPaccum5
DPaccum1			Max2	FIRaccum1
DPaccum0			Max1	FFTaccum2
			Max0	Max6
				DPaccum4

creating a merged process of the merged thread

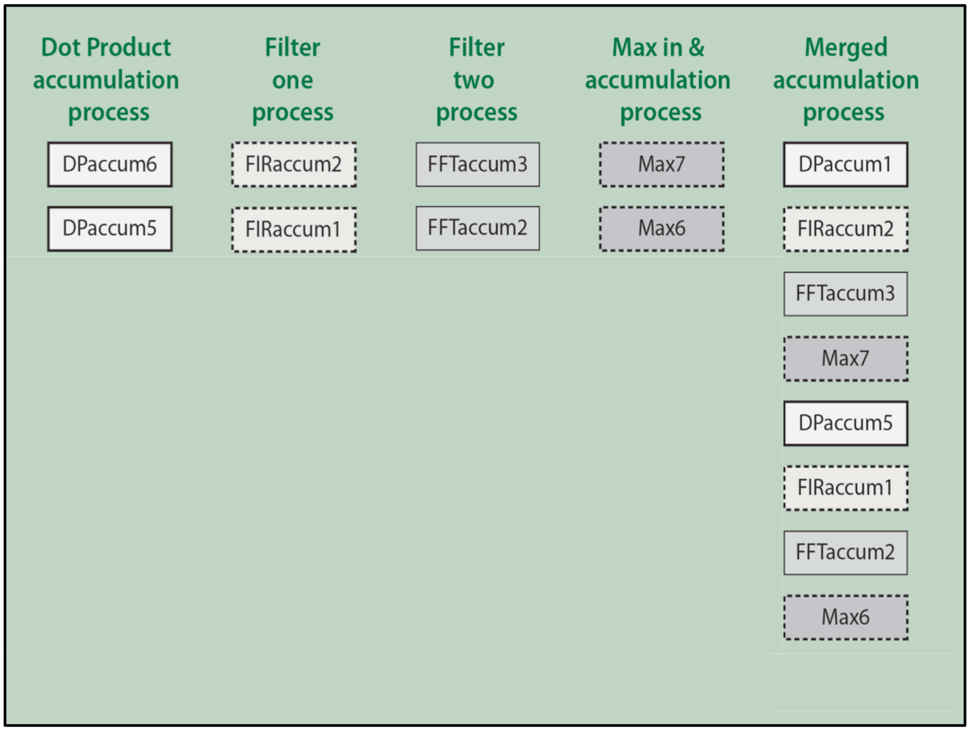


Dot Product accumulation process	Filter one process	Filter two process	Max in & accumulation process	Merged accumulation process
DPaccum6	FIRaccum2	FFTaccum3	Max7	

The highest priority, process state is the top state. It has minimal probability. Merging processes means

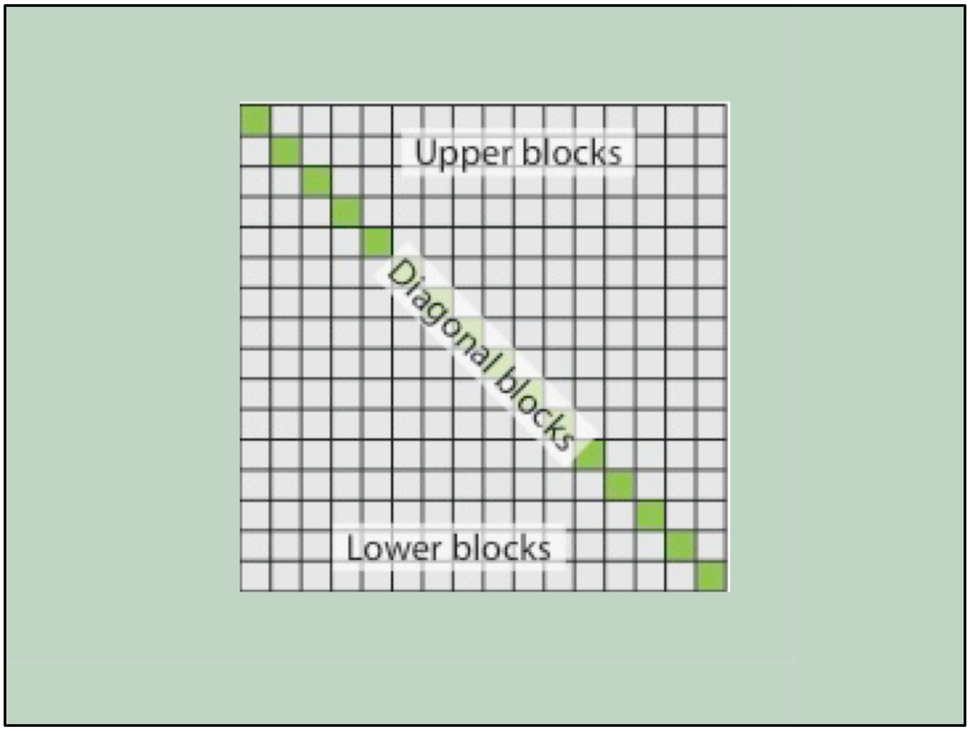


listing the highest priority states in the merged process.



Dot Product accumulation process	Filter one process	Filter two process	Max in & accumulation process	Merged accumulation process
DPaccum6	FIRaccum2	FFTaccum3	Max7	DPaccum1
DPaccum5	FIRaccum1	FFTaccum2	Max6	FIRaccum2
				FFTaccum3
				Max7
				DPaccum5
				FIRaccum1
				FFTaccum2
				Max6

Merging continues in this way.



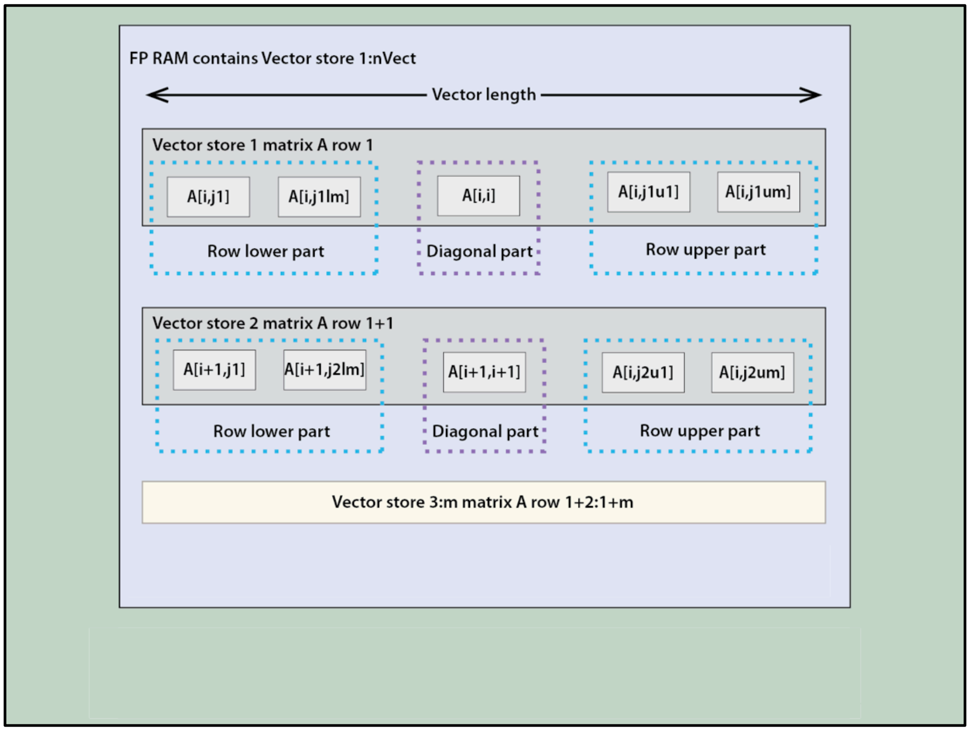
This shows thread placement of a dense, matrix algorithm
as lower, diagonal and upper block threads.

Here, the integer and Pat Man cores are turned off.

The DPC has a 12 by 12 array of PEM, each including 4 FP cores with dual fused Multiplier C-Adders.

There are 1152 fused Multiplier C-Adders of 64 by 64, or dual 32 by 32, or quad 16 by 16 FP or Posit arithmetic, generating results every ns.

Dense matrix by matrix operations, can proceed at the rate of the multipliers.



FP RAM contains Vector store 1:nVect

Vector length

Vector store 1 matrix A row 1

A[i,j1]

A[i,j1lm]

A[i,i]

A[i,j1u1]

A[i,j1um]

Row lower part

Diagonal part

Row upper part

Vector store 2 matrix A row 1+1

A[i+1,j1]

A[i+1,j2lm]

A[i+1,i+1]

A[i,j2u1]

A[i,j2um]

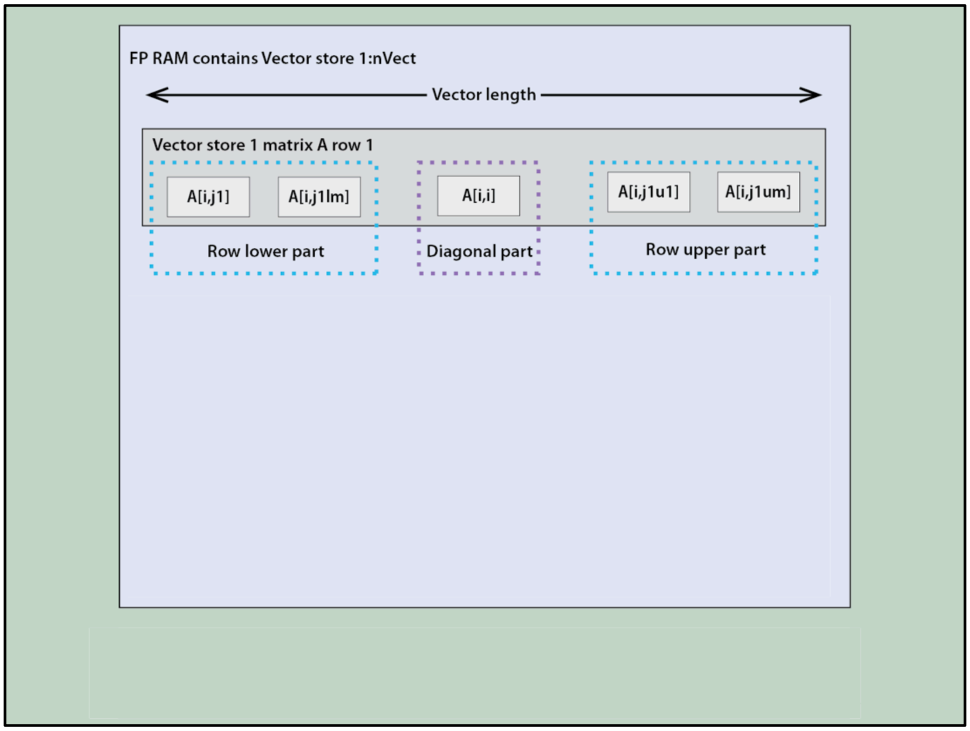
Row lower part

Diagonal part

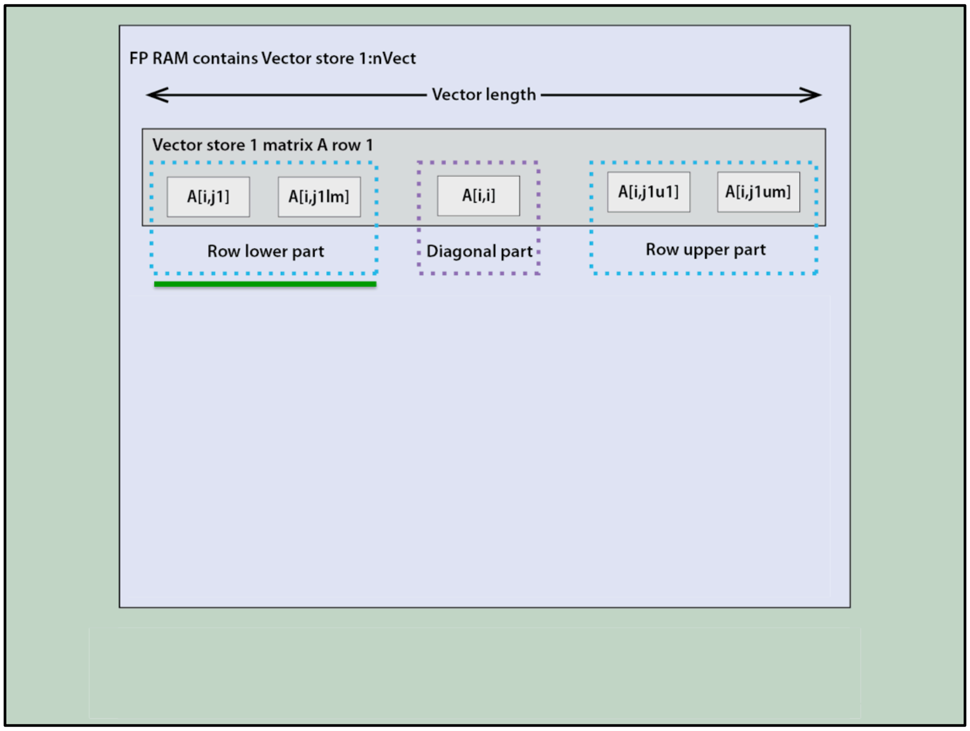
Row upper part

Vector store 3:m matrix A row 1+2:1+m

Let’s briefly discuss sparse matrix manipulation in the core module.
Each core module often uses all the FP RAM, and at least ½ of the Integer RAM.
The FP RAM holds all the non-zero entries for several rows of the Matrix (A).



Each of these row entries includes


FP RAM contains Vector store 1:nVect

Vector length

Vector store 1 matrix A row 1

A[i,j1]

A[i,j1lm]

A[i,i]

A[i,j1u1]

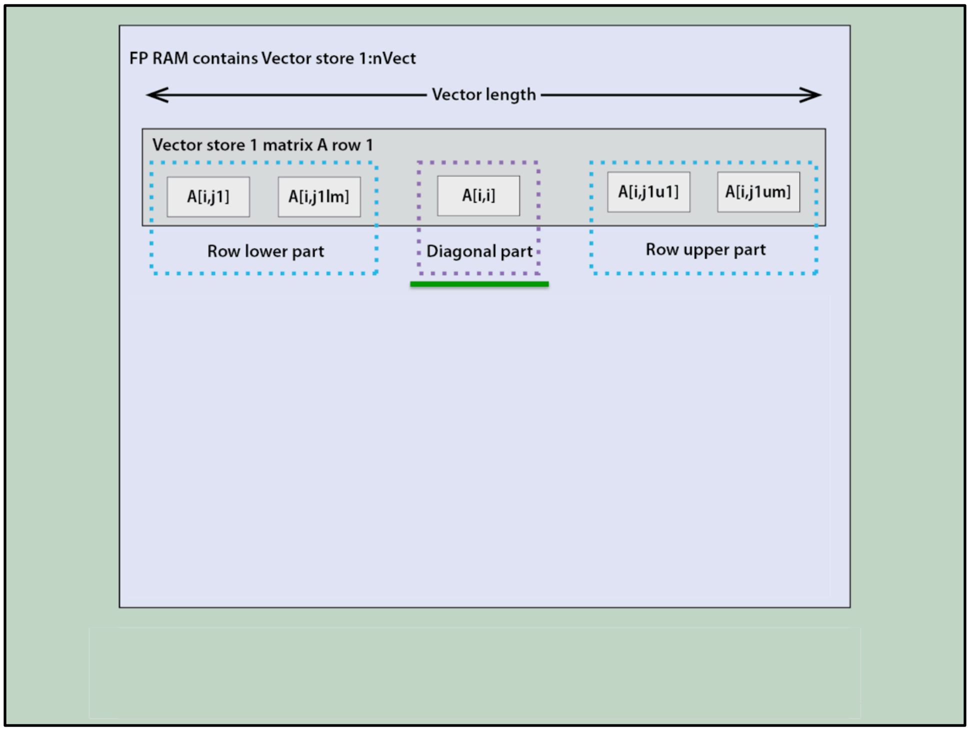
A[i,j1um]

Row lower part

Diagonal part

Row upper part

A lower part



FP RAM contains Vector store 1:nVect

Vector length

Vector store 1 matrix A row 1

A[i,j1]

A[i,j1lm]

A[i,i]

A[i,j1u1]

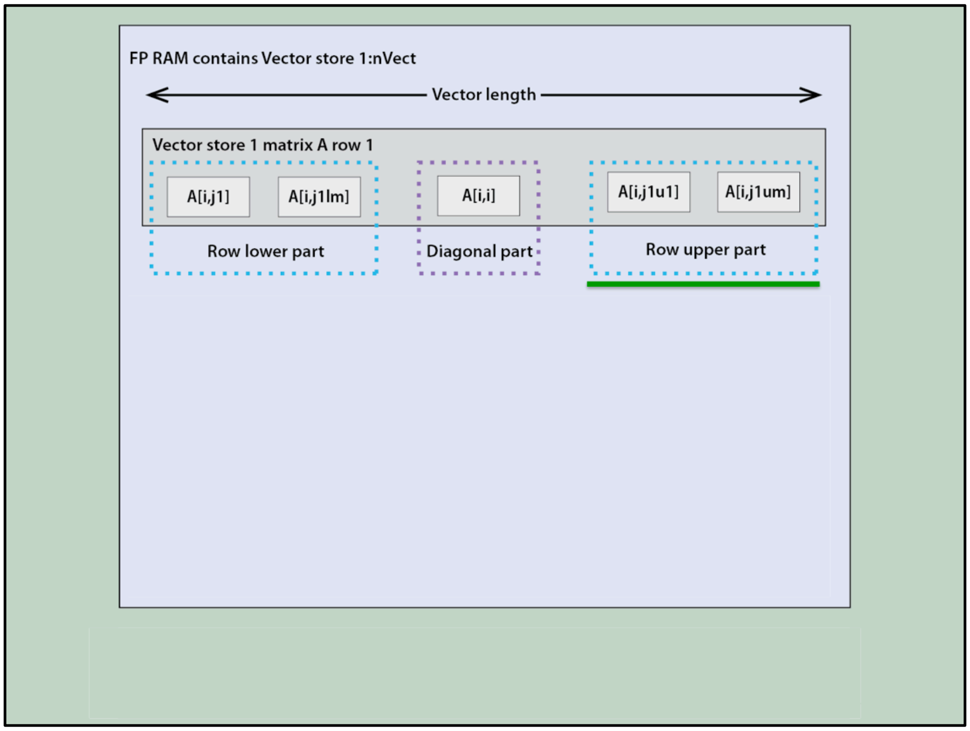
A[i,j1um]

Row lower part

Diagonal part

Row upper part

A diagonal part

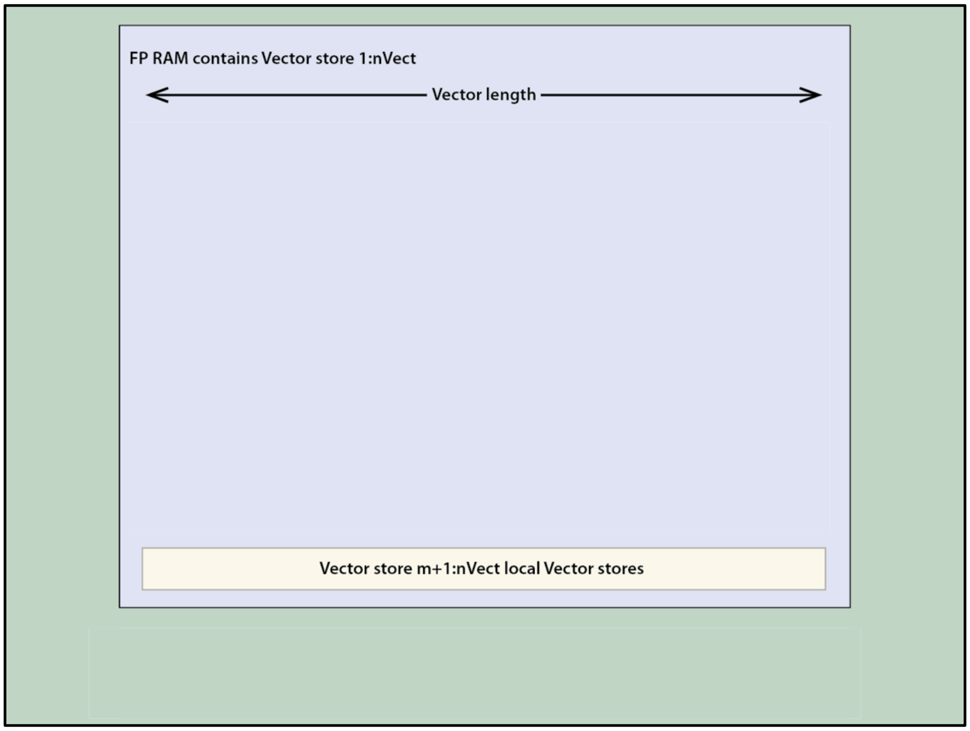


FP RAM contains Vector store 1:nVect

Vector length

Vector store 1 matrix A row 1

And an upper part.

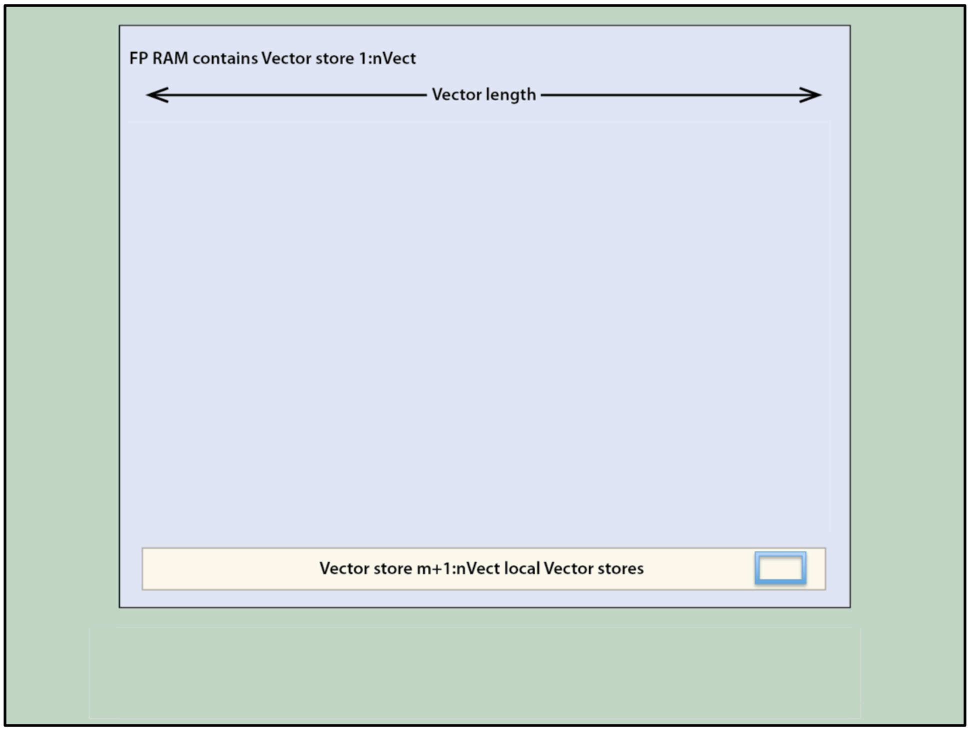


FP RAM contains Vector store 1:nVect

Vector length

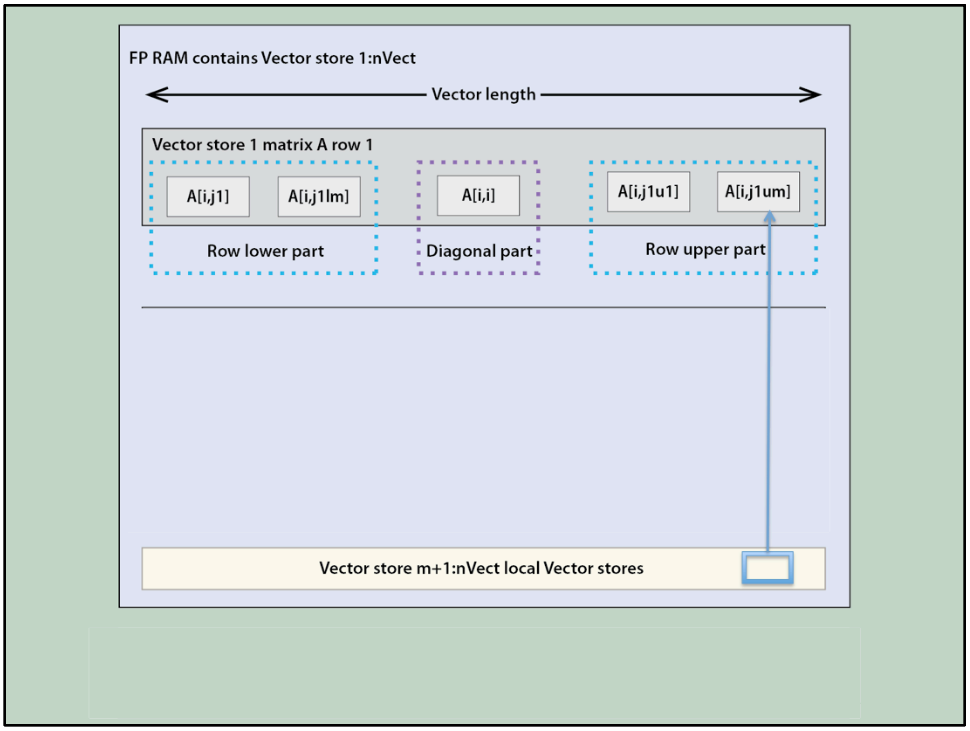
Vector store m+1:nVect local Vector stores

The relevant entries of each vector are stored locally in vector stores in the FP Ram.

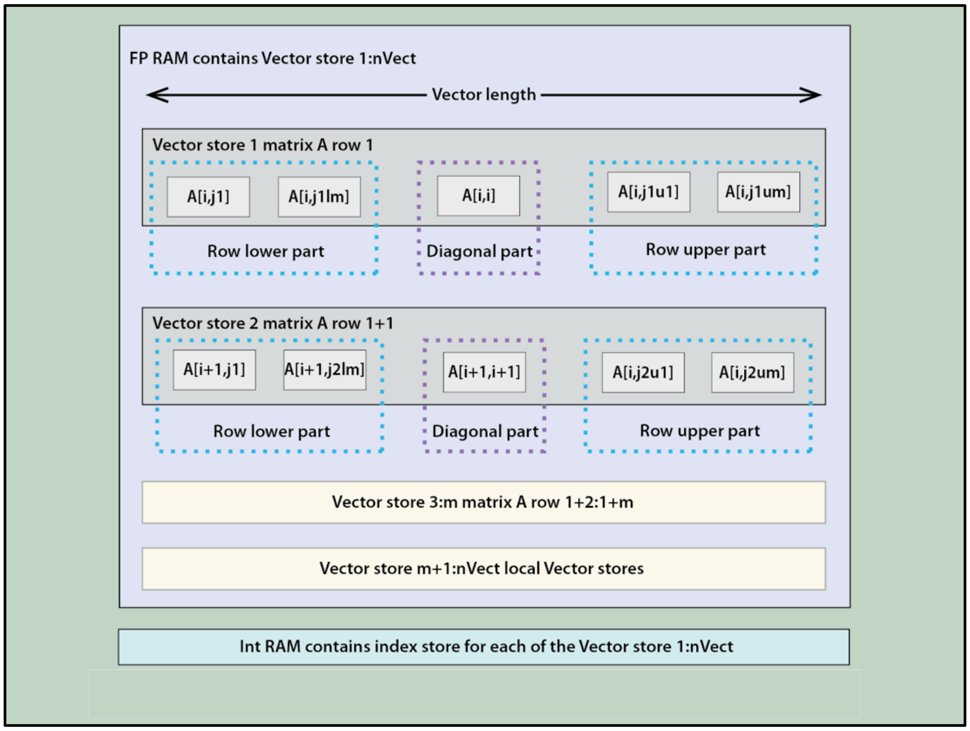


FP RAM contains Vector store 1:nVect
Vector length
Vector store m+1:nVect local Vector stores

Each of these entries


FP RAM contains Vector store 1:nVect
Vector length
Vector store 1 matrix A row 1
A[i,j1]
A[i,j1lm]
A[i,i]
A[i,j1u1]
A[i,j1um]
Row lower part
Diagonal part
Row upper part
Vector store m+1:nVect local Vector stores

correspond to at least one non-zero A row entry.



FP RAM contains Vector store 1:nVect

Vector length

Vector store 1 matrix A row 1

A[i,j1] A[i,j1lm] A[i,i] A[i,j1u1] A[i,j1um]

Row lower part

Diagonal part

Row upper part

Vector store 2 matrix A row 1+1

A[i+1,j1] A[i+1,j2lm] A[i+1,i+1] A[i,j2u1] A[i,j2um]

Row lower part

Diagonal part

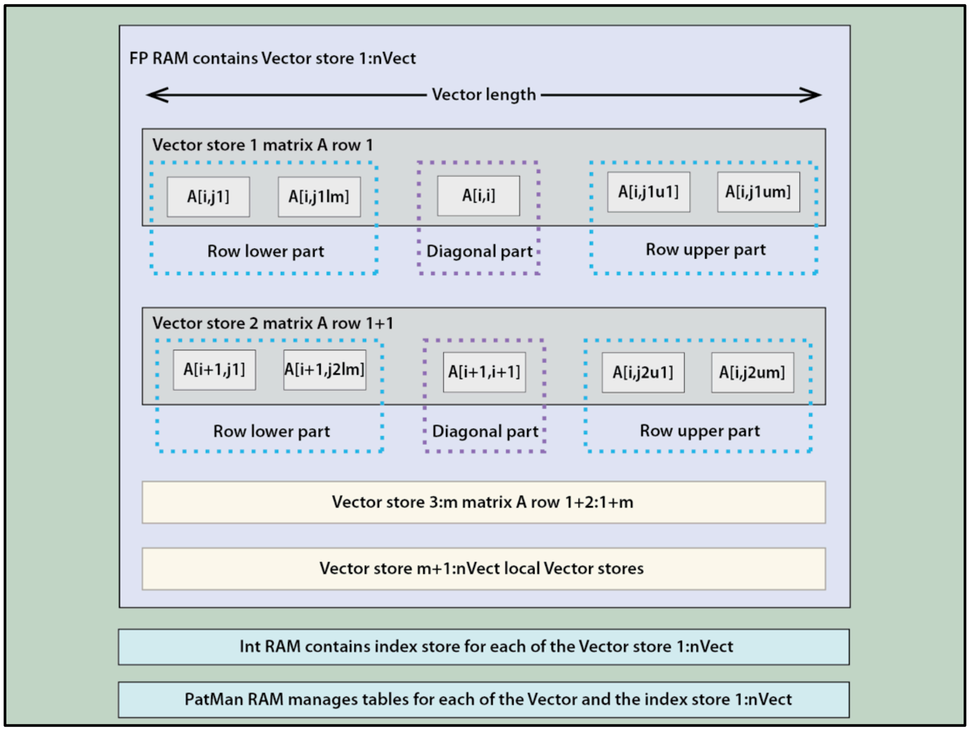
Row upper part

Vector store 3:m matrix A row 1+2:1+m

Vector store m+1:nVect local Vector stores

Int RAM contains index store for each of the Vector store 1:nVect

Integer RAM holds the relevant indices for the corresponding entries of the FP RAM



FP RAM contains Vector store 1:nVect

Vector length

Vector store 1 matrix A row 1

A[i,j1]

A[i,j1lm]

Row lower part

A[i,i]

Diagonal part

A[i,j1u1]

A[i,j1um]

Row upper part

Vector store 2 matrix A row 1+1

A[i+1,j1]

A[i+1,j2lm]

Row lower part

A[i+1,i+1]

Diagonal part

A[i,j2u1]

A[i,j2um]

Row upper part

Vector store 3:m matrix A row 1+2:1+m

Vector store m+1:nVect local Vector stores

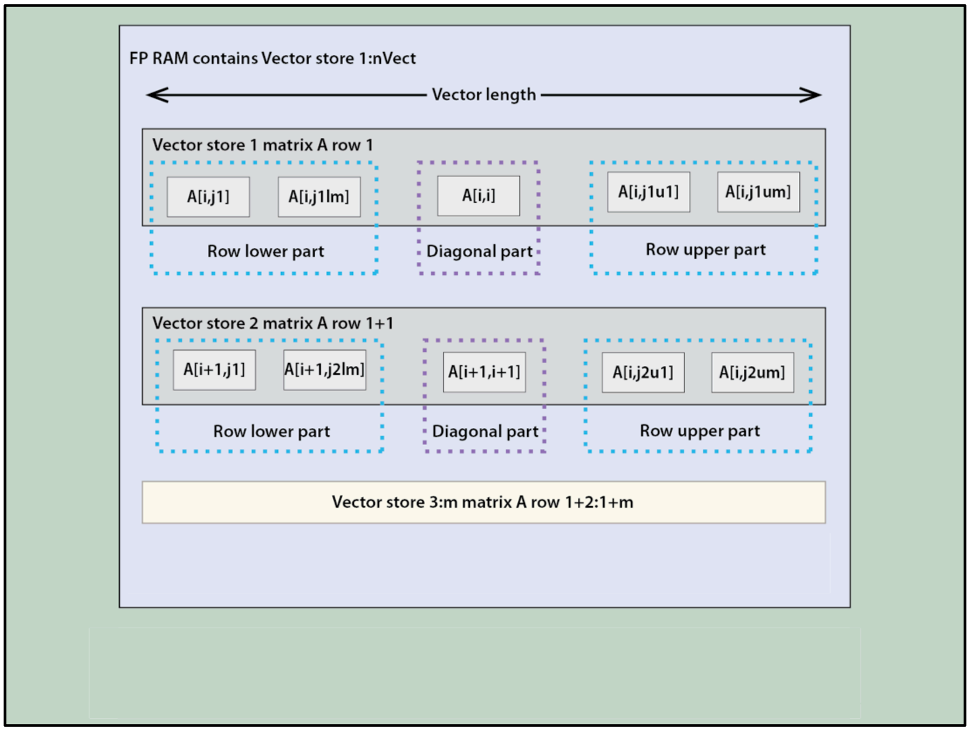
Int RAM contains index store for each of the Vector store 1:nVect

PatMan RAM manages tables for each of the Vector and the index store 1:nVect

Pat Man RAM contains the management tables for the FP and Int cores

FP RAM contains Vector store 1:nVect
Vector length

Consider HPCG:
A vector store length
of 27 can hold
the body and boundary components of the A matrix rows.
Making all the stores of the FP Ram this same size, removes the need for garbage collection.



FP RAM contains Vector store 1:nVect

Vector length

Vector store 1 matrix A row 1

A[i,j1] A[i,j1lm] A[i,i] A[i,j1u1] A[i,j1um]

Row lower part

Diagonal part

Row upper part

Vector store 2 matrix A row 1+1

A[i+1,j1] A[i+1,j2lm] A[i+1,i+1] A[i,j2u1] A[i,j2um]

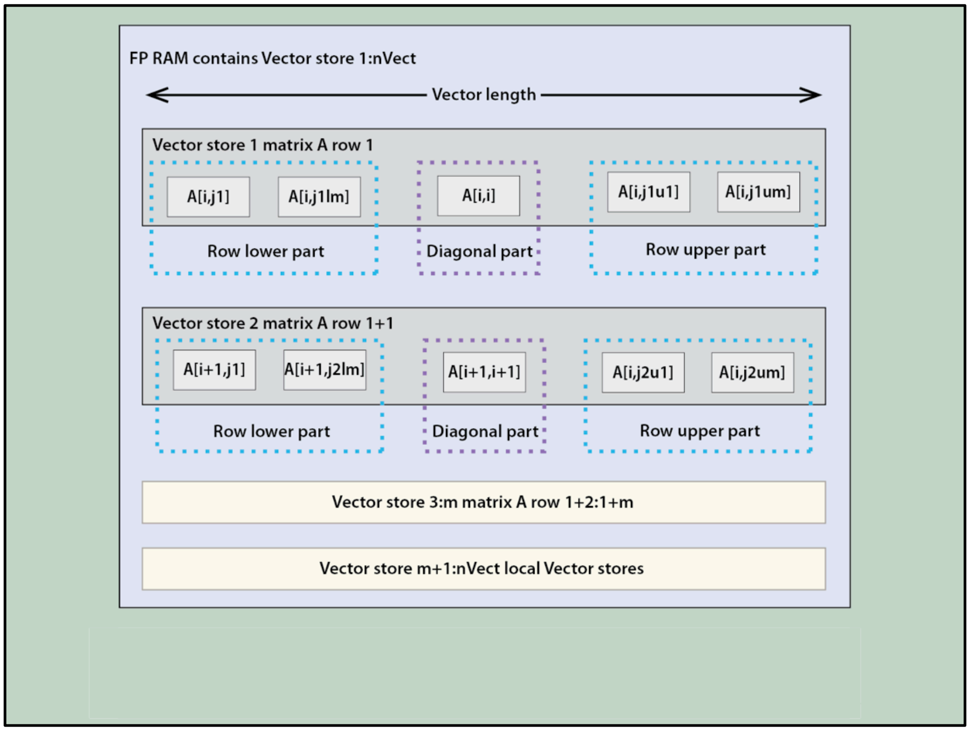
Row lower part

Diagonal part

Row upper part

Vector store 3:m matrix A row 1+2:1+m

Each FP RAM of 4 K words supports 151 vector stores.



FP RAM contains Vector store 1:nVect

Vector length

Vector store 1 matrix A row 1

A[i,j1] A[i,j1lm] A[i,i] A[i,j1u1] A[i,j1um]

Row lower part

Diagonal part

Row upper part

Vector store 2 matrix A row 1+1

A[i+1,j1] A[i+1,j2lm] A[i+1,i+1] A[i,j2u1] A[i,j2um]

Row lower part

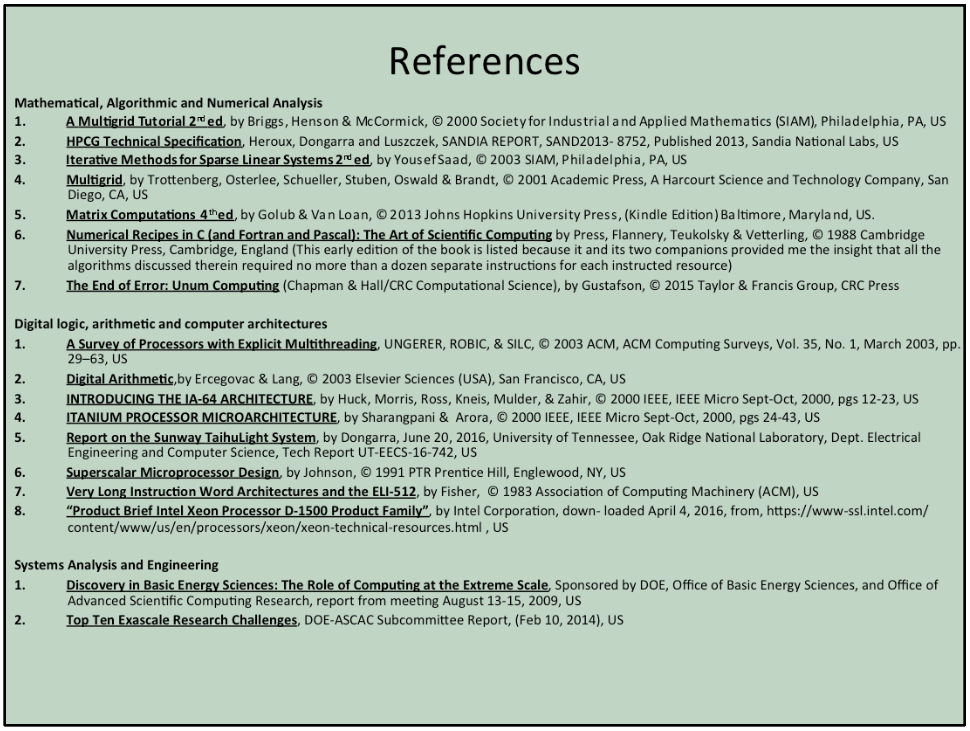
Diagonal part

Row upper part

Vector store 3:m matrix A row 1+2:1+m

Vector store m+1:nVect local Vector stores

Assume that 25% of these stores are committed to the local components of the vectors.
Then each core module can execute about 110 rows.
The only overhead is communication of vector updates, most of which are local to the DPC.
The 48,384,000 rows of HPCG fit in 440 K PEM or about 744 DPC, filling about 1 rack.
The model never needs to leave the PEM. DRAM is read only once.



References

Mathematical, Algorithmic and Numerical Analysis

1. **A Multigrid Tutorial 2nd ed**, by Briggs, Henson & McCormick, © 2000 Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, US
2. **HPCG Technical Specification**, Heroux, Dongarra and Luszczek, SANDIA REPORT, SAND2013- 8752, Published 2013, Sandia National Labs, US
3. **Iterative Methods for Sparse Linear Systems 2nd ed**, by Yousef Saad, © 2003 SIAM, Philadelphia, PA, US
4. **Multigrid**, by Trottenberg, Osterlee, Schueller, Stuben, Oswald & Brandt, © 2001 Academic Press, A Harcourt Science and Technology Company, San Diego, CA, US
5. **Matrix Computations 4th ed**, by Golub & Van Loan, © 2013 Johns Hopkins University Press, (Kindle Edition) Baltimore, Maryland, US.
6. **Numerical Recipes in C (and Fortran and Pascal): The Art of Scientific Computing** by Press, Flannery, Teukolsky & Vetterling, © 1988 Cambridge University Press, Cambridge, England (This early edition of the book is listed because it and its two companions provided me the insight that all the algorithms discussed therein required no more than a dozen separate instructions for each instructed resource)
7. **The End of Error: Unum Computing** (Chapman & Hall/CRC Computational Science), by Gustafson, © 2015 Taylor & Francis Group, CRC Press

Digital logic, arithmetic and computer architectures

1. **A Survey of Processors with Explicit Multithreading**, UNGERER, ROBIC, & SILC, © 2003 ACM, ACM Computing Surveys, Vol. 35, No. 1, March 2003, pp. 29–63, US
2. **Digital Arithmetic**,by Ercegovac & Lang, © 2003 Elsevier Sciences (USA), San Francisco, CA, US
3. **INTRODUCING THE IA-64 ARCHITECTURE**, by Huck, Morris, Ross, Kneis, Mulder, & Zahir, © 2000 IEEE, IEEE Micro Sept-Oct, 2000, pgs 12-23, US
4. **ITANIUM PROCESSOR MICROARCHITECTURE**, by Sharangpani & Arora, © 2000 IEEE, IEEE Micro Sept-Oct, 2000, pgs 24-43, US
5. **Report on the Sunway TaihuLight System**, by Dongarra, June 20, 2016, University of Tennessee, Oak Ridge National Laboratory, Dept. Electrical Engineering and Computer Science, Tech Report UT-EECS-16-742, US
6. **Superscalar Microprocessor Design**, by Johnson, © 1991 PTR Prentice Hill, Englewood, NY, US
7. **Very Long Instruction Word Architectures and the ELI-512**, by Fisher, © 1983 Association of Computing Machinery (ACM), US
8. **"Product Brief Intel Xeon Processor D-1500 Product Family"**, by Intel Corporation, down- loaded April 4, 2016, from, https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-technical-resources.html , US

Systems Analysis and Engineering

1. **Discovery in Basic Energy Sciences: The Role of Computing at the Extreme Scale**, Sponsored by DOE, Office of Basic Energy Sciences, and Office of Advanced Scientific Computing Research, report from meeting August 13-15, 2009, US
2. **Top Ten Exascale Research Challenges**, DOE-ASCAC Subcommittee Report, (Feb 10, 2014), US

Thank you. Are there any questions?