CSc 131
Computer Software Engineering

Chapter 3
Computer Taxonomy

Herbert G. Mayer, CSU CSC

Status 8/15/2019

Syllabus

- Processor Introductions
- Common Architecture Attributes
- Stream Classification Flynn 1966
- Generic Computer Architecture Model
- Instruction Set Architecture (ISA)
- Iron Law of Performance
- Uniprocessor (UP) Architectures
- Multiprocessor (MP) Architectures
- Hybrid Architectures
- References

Computer Taxonomy

Introduction Uniprocessors

- Single Accumulator Architecture (earliest systems 1940s), e.g. John von Neumann's computer, or John Vincent Atanasoff's computer
  - Was basis for ENIAC
- General-Purpose Register Architectures (GPR)
- 2-Address Architecture (GPR with one operand implied), e.g. IBM 360
- 3-Address Architecture (GPR with all operands of arithmetic operation explicit), e.g. VAX 11/70
- Stack Machines (e.g. B5000 see [2], B6000, HP3000 see [3])
- Vector Architecture, e.g. Amdahl 470/6, competing with IBM's 360 in the 1970s; blurs line to Multiprocessor

Introduction Multiprocessors

- Shared Memory Architecture
- Distributed Memory Architecture
- Systolic Architecture; see Intel® iWarp and CMU's warp architecture
- Data Flow Machine; see Jack Dennis' work at MIT

Introduction Hybrid Processors

- Superscalar Architecture; see Intel 80860, AKA i860
- VLIW Architecture; see Multiflow computer
- Pipelined Architecture; debatable ☺ whether UP or hybrid
- EPIC Architecture; see Intel® Itanium® architecture (now EOL-ed)

Common Architecture Attributes

- Main memory (main store), separate from CPU
- Program instructions stored in main memory
- Also, data stored in memory; aka von Neumann architecture
- Data available in –distributed over– main memory, stack, heap, OS space, free space, IO space
- Instruction pointer (AKA instruction counter, program counter), other special registers
- Von Neumann memory bottle-neck, everything travels on same bus
- Accumulator (register, 1 or many) holds result of arithmetic/logical operation

Common Architecture Attributes

- IO Controller handles memory access requests from processor, to memory; AKA “chipset”
- Current trend is to move all or part of memory controller onto CPU chip; no, that does not mean the controller IS part of the CPU!
- Processor units include: FP units, Integer unit, control unit, register file, pathways

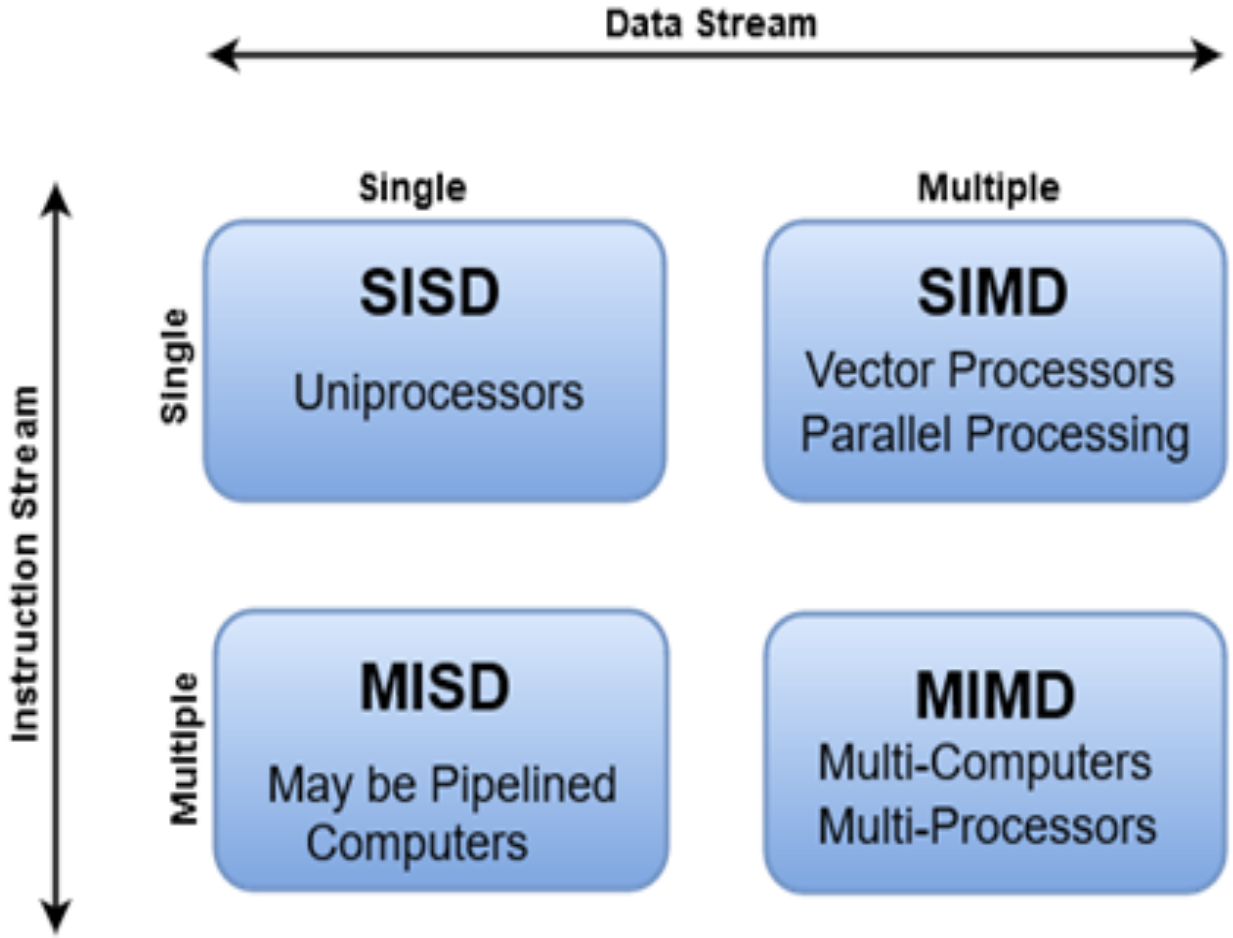
Data-Stream, Instruction-Stream

The common four Data-Stream, Instruction-Stream classifications defined by Michael J. Flynn 1966!

- **Single-Instruction, Single-Data Stream (SISD) Architecture, e.g. (PDP-11)**
- **Single-Instruction, Multiple-Data Stream (SIMD) Architecture, e.g. Array Processors, Solomon, Illiac IV, BSP, TMC**
- **Multiple-Instruction, Single-Data Stream (MISD) Architecture, e.g. possibly: superscalar machines, pipelined, VLIW, EPIC**
- **Multiple-Instruction, Multiple-Data Stream Architecture (MIMD); perhaps true multiprocessor**

Data-Stream, Instruction-Stream

Flynn's Classification of Computers



Nature of Data

- **Spreadsheets give us a way to create processes that manipulate data**
- **We can think of a piece of data as a quantity that can be stored in a cell of a worksheet**
- **That cell represents an identifiable, physical location where the data resides**
- **What happens when you work on a spreadsheet or other program**

Computer Organization

A typical computer is composed of the following common basic modules:

- **Processor: "chip" where the work gets done**
- **Might be an Intel chip or AMD or some other kind. Modern chips have multiple processors ("cores") on same piece of silicon that can share work**
- **Modern computers have yet another processor for graphics processing**
- **A (mis) measure of chip quality is speed: GHz**
- **Amount of heat (in Watts) generated is critical**

Computer Organization

- **Memory:** Like memory on your microwave oven ☺ when power is off, memory loses its contents
- Memory relatively fast, connected closely with CPU, but memory speed is slow vs. CPU clock speed
- Memory speed often measured in Megaherz; faster is better
- Size of memory is measured in Gigabytes and more; today 64-bit addresses
- Storage, implemented as: disks, SSDs, or flash drives
- Flash storage holds value when power is off, but is slower than regular memory. Size ~ Gigabytes

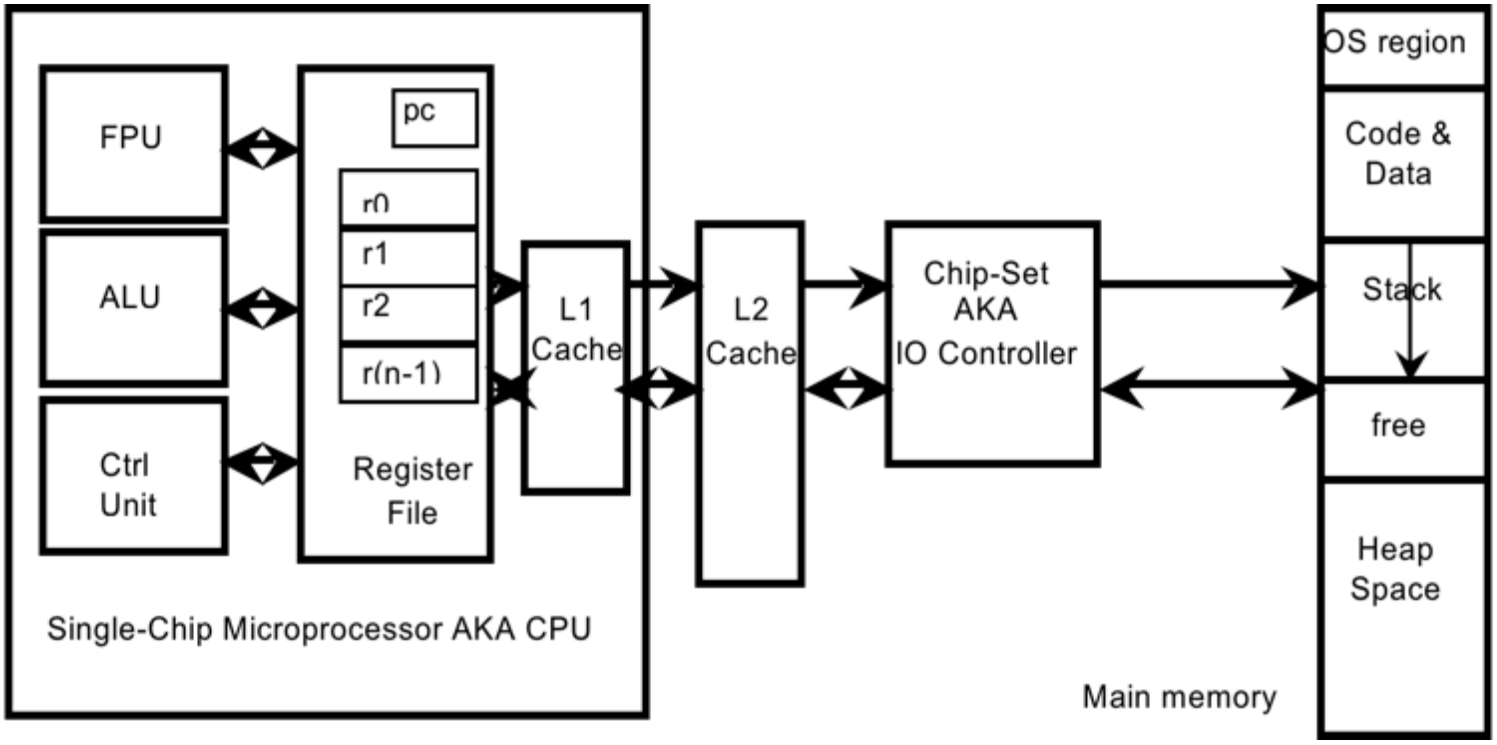
Computer Organization

- **Cache: This is a special-purpose memory that mediates between working memory and the processor. Having a good-size cache can make the computer significantly faster**
- **Caches are fast, yet expensive, consume real power!**
- **Input devices: Keyboard, mouse, and microphone but also game controllers, laboratory instruments, etc.**
- **Output devices, such as screen, printer, speaker, and others, like robot arms or a car's brake system**

What Constitutes a Computer?

- Many modern appliances have built-in chips; e.g. cars, televisions, washing machines, etc.
- What makes a computer special is that it stores its program in memory, just like data, and can accept and execute new programs
- It can also potentially modify its own stored programs ("learn")

Computer Modules



Bits

- **Bit: Short for Binary digit**
- **Everything in the computer, whether it is text, numerical data, music, videos, programs like Excel, macros or other code you write, or the operating system (Windows, MacOS, OSX, Linux) is stored digitally in these bits, tiny devices ☺ that have two distinct physical states**
- **Abstractly these states are represented by the numbers 0 and 1. Such an individual 0 or 1 container is the bit**

Moving bits, Simplistic Model

- It would be too inefficient for the computer to access and move bits individually, though bit-addressed computers have been designed, e.g. NCR successor of Century series
- Bits are grouped into chunks that are moved around the computer, from the disk to the memory to the CPU, as a group
- The bigger the chunk is, the faster the computer can move data. A “32 bit” computer moves data in 32 bit chunks. A “64 bit” computer moves data in bigger 64-bit chunks and can conveniently handle a bigger working memory
- The original chunk was typically a byte (8 bits)
- Note that bus width and natural data width are orthogonal

Operating System

- The operating system (AKA "OS") is *the* key system SW on your computer: it controls what happens, and manages resources
- If you have multiple programs open, for example, and you type something, it figures out where the characters should go
- While the computer is up and running, the operating system is resident in working memory, to be fast
- But when the computer is turned off … all memory information is wiped out
- So when you turn your computer on, how does it get started with no operating system?

Booting

- A special, dedicated device = chip that retains its data when power is off, has a small program built in
- When you turn on your computer this program loads the actual operating system from external storage into the working memory
- This is called “booting” because it is supposedly like picking yourself up by your own bootstraps
- This is what is going on during the minutes (on Windows) after you turn on the computer before your operating system finally pops up

Applications

- **Besides your operating system and some associated programs, your computer also has application programs that actually do the things you want to do:**
- **E.g. word processing, games, spreadsheets, play music, computations, etc.**
- **The typical computer user just *uses* applications**

Cray 2 Supercomputer

Instruction Set Architecture (ISA)

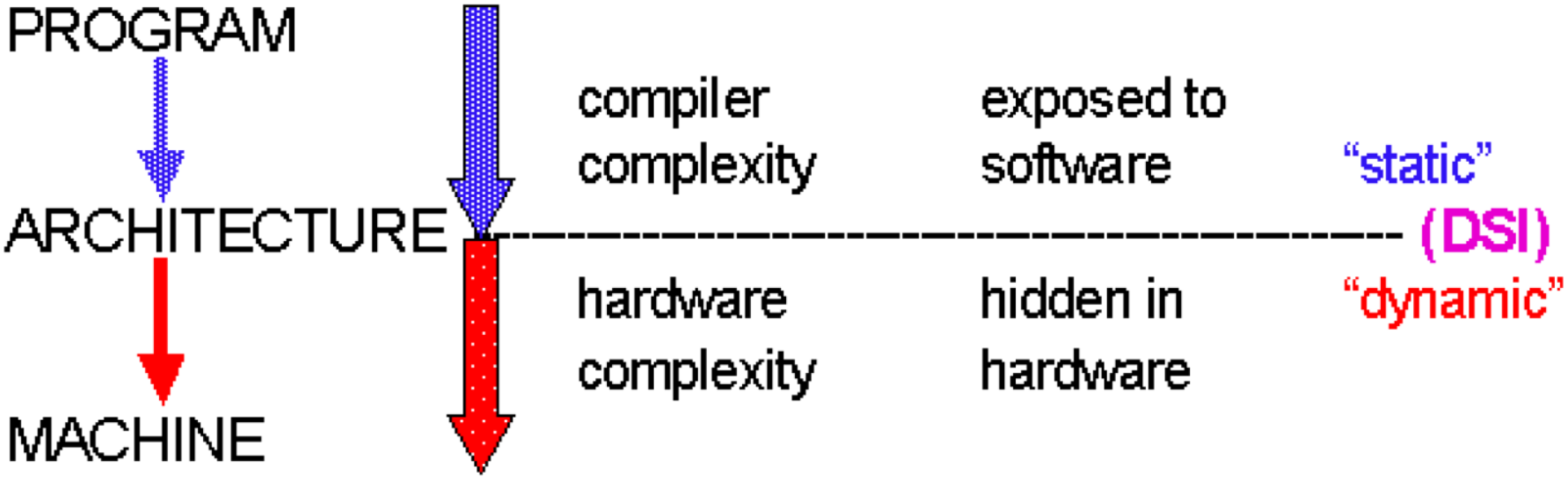
- ***ISA is boundary between Software (SW) and Hardware (HW)***
- **Specifies logical machine that is visible to the programmer & compiler**
- **ISA is the functional specification for processor designers**
- **That boundary is sometimes a very low-level piece of system software that handles exceptions, interrupts, and HW-specific services**
- **Could fall into the domain of the OS**

Instruction Set Architecture (ISA)

- *Specified by an ISA:*
- Operations: what to perform and in which order
- Temporary Operand Storage in CPU: accumulator, stack, registers
- Note that stack can be word-sized, even bit-sized (design of successor for NCR's Century architecture of the 1970s)
- Number of operands per instruction, 0, 1, 2, more
- Operand location: where and how to place or locate any operands
- Type and size of operands
- Instruction Encoding in binary

Instruction Set Architecture (ISA)

ISA: Dynamic Static Interface (DSI)



Iron Law of Processor Performance

- Clock-rate doesn't count, bus width doesn't count, the number of registers and operations executed in parallel doesn't count! ☺
- What counts is how long it takes *for my computational task to complete*. That time is of the essence of computing!
- If a MIPS-based solution runs at 1 GHz, completing a program X in 2 minutes, while an Intel Pentium® 4–based program runs at 3 GHz and completes that same program x in 2.5 minutes, programmers/users are more interested in the former solution

Iron Law of Processor Performance

Iron Law of Processor Performance

- If a solution on an Intel CPU can be expressed in an object program of size Y bytes, but on an IBM architecture of size 1.1 Y bytes, the Intel solution is generally more attractive
- Assuming same execution, performance
- Meaning of this:
  - *Wall-clock time* (*Time*) is time I have to wait for completion
  - *Program Size* is indicator of overall physical complexity of computational task

Iron Law of Processor Performance

$$1 \,/\, \text{Processor Performance} = \frac{\text{Time}}{\text{Program Size}}$$

$$= \frac{\text{Instructions}}{\text{Program Size}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

(code size)	*(Cpi)*	*(cycle time)*
→	→	→
Architecture	Implementation	Realization
Compiler Designer	Processor Designer	Actual Chip

Computing History

Computing Before Computers

- Babylonian number system based on 60: Today's convention of 60 sec. per minute, and 60 min. per hour are a left over of that convention
- Roman numerals; usable but not practical for real computing; lacked even a million, expressed as "a thousand thousands"
- Abacus used in much of Asia, even today, with base of 10 and powers of 10; though 10 split into 2 groups of 5
- Arabic number system with base 10, powers of 10 by position, and introduction of the 0; made computing practical

Computing History

Long, long before 1940s:

1643 Pascal's *Arithmetic Machine*

About 1660 Leibnitz *Four Function Calculator*

1710 -1750 *Punched Cards* by Bouchon, Falcon, Jacquard

1810 Babbage *Difference Engine*, unfinished; 1st programmer ever in the world was Ada, poet Lord Byron's daughter, after whom the language Ada was named: Lady Ada Lovelace

1835 Babbage *Analytical Engine*, also unfinished

1920 Hollerith *Tabulating Machine*

to help with census in the USA

Library of Congress

Computing History

Decade of 1940s

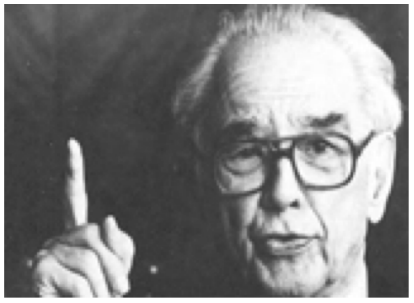
1939 – 1942 John Atanasoff built programmable, electronic computer at Iowa State University

1936 - 1945 Konrad Zuse's Z3 and Z4, early electro-mechanical computers based on relays; colleague advised use of "vacuum tubes"; WW-2 cut short his component supply

1946 John von Neumann's computer design of stored program

1946 Mauchly and Eckert built *ENIAC*, modeled after Atanasoff's ideas, built at University of Pennsylvania: Electronic Numeric Integrator and Computer, 30 ton monster

1980s John Atanasoff: official acknowledgment & patent; no $



Computing History

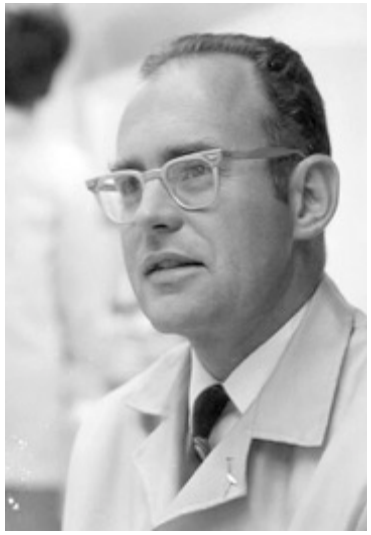
Decade of the 1950s

- Univac Uniprocessor based on ENIAC, commercially viable, developed by John Mauchly and John Presper Eckert
- Commercial systems sold by Remington Rand
- Mark III computer

Decade of the 1960s

- IBM's 360 family co-developed with GE, Siemens, et al.
- Transistor replaces vacuum tube
- Burroughs stack machines, compete with GPR architectures
- All still von Neumann architectures
- 1969 *ARPANET*
- *Cache* and *VMM* developed, at Manchester University UK

Computing History



Decade of the 1970s

- Birth of Microprocessor at Intel,
  see Gordon Moore
- High-end mainframes, e.g. CDC 6000s, IBM 360 + 370 series
- Architecture advances: Caches, virtual memories (VMM) ubiquitous, since real memories were expensive
- Intel 4004, Intel 8080, single-chip microprocessors
- Programmable controllers
- Mini-computers, PDP 11, HP 3000 16-bit computer
- Height of Digital Equipment Corp. (DEC)
- Birth of personal computers, which DEC misses!

Computing History

Decade of the 1980s

decrease of mini-computer use

32-bit computing even on minis

Architecture advances:

- superscalar, faster caches, larger caches

Multitude of Supercomputer manufacturers

Compiler complexity: trace-scheduling, VLIW

Workstations common: Apollo, HP, DEC's Ken Olsen trying to catch up, Intergraph, Ardent, Sun, Three Rivers, Silicon Graphics, etc.

Computing History

Decade of the 1990s

- Architecture advances: superscalar & pipelined, speculative execution, ooo execution
- Powerful desktops
- End of mini-computer and of many super-computer manufacturers
- Microprocessor powerful as early supercomputers
- Consolidation of many computer companies into few larger ones
- End of USSR marked the demise of numerous computer- and supercomputer companies

Evolution of *μ*P Performance (by: James C. Hoe @ CMU)

	1970s	1980s	1990s	2000+
Transistor Count	10k-100k	100k-1M	1M-100M	1B
Clock Frequency	0.2-2 MHz	2-20 MHz	0.02 – 1 GHz	10 GHz
Instructions / cycle: ipc	< 0.1	0.1 – 0.9	0.9 – 2.0	> 10 (?)
MIPs, FLOPs	< 0.2	0.2 - 20	20 – 2,000	100,000

Processor Performance Growth

Moore's Law --from Webopedia:

- "The observation made in 1965 by Gordon Moore, co-founder of Intel, that the number of transistors per square inch on integrated circuits had doubled every year since it was invented. Moore predicted that this trend would continue for the foreseeable future.
- In subsequent years, the pace slowed down a bit, but ***data density doubled approximately every 18 months***, and this is the current definition of *Moore's Law*, which Moore himself has blessed. Most experts, including Moore himself, expect *Moore's Law* to hold for another two decades.
- Others coin a more general law, a bit lamely stating that *"the circuit density increases predictably over time."*

Processor Performance Growth

- In the 2010s, Moore's Law holding true since ~1968
- Some Intel *fellows* believed, an end to Moore's Law would be reached ~2018 due to physical limitations in the process of manufacturing transistors from semi-conductor material
- Such phenomenal growth is unknown in any other industry: If doubling of performance could be achieved every 18 months, then by early 2000s other industries would have achieved the following:
  - Cars would travel at 2,400,000 Mph, and get 600,000 MpG
  - Air travel LA to NYC would be at 36,000 Mach, take 0.5 seconds

Key Architecture Messages

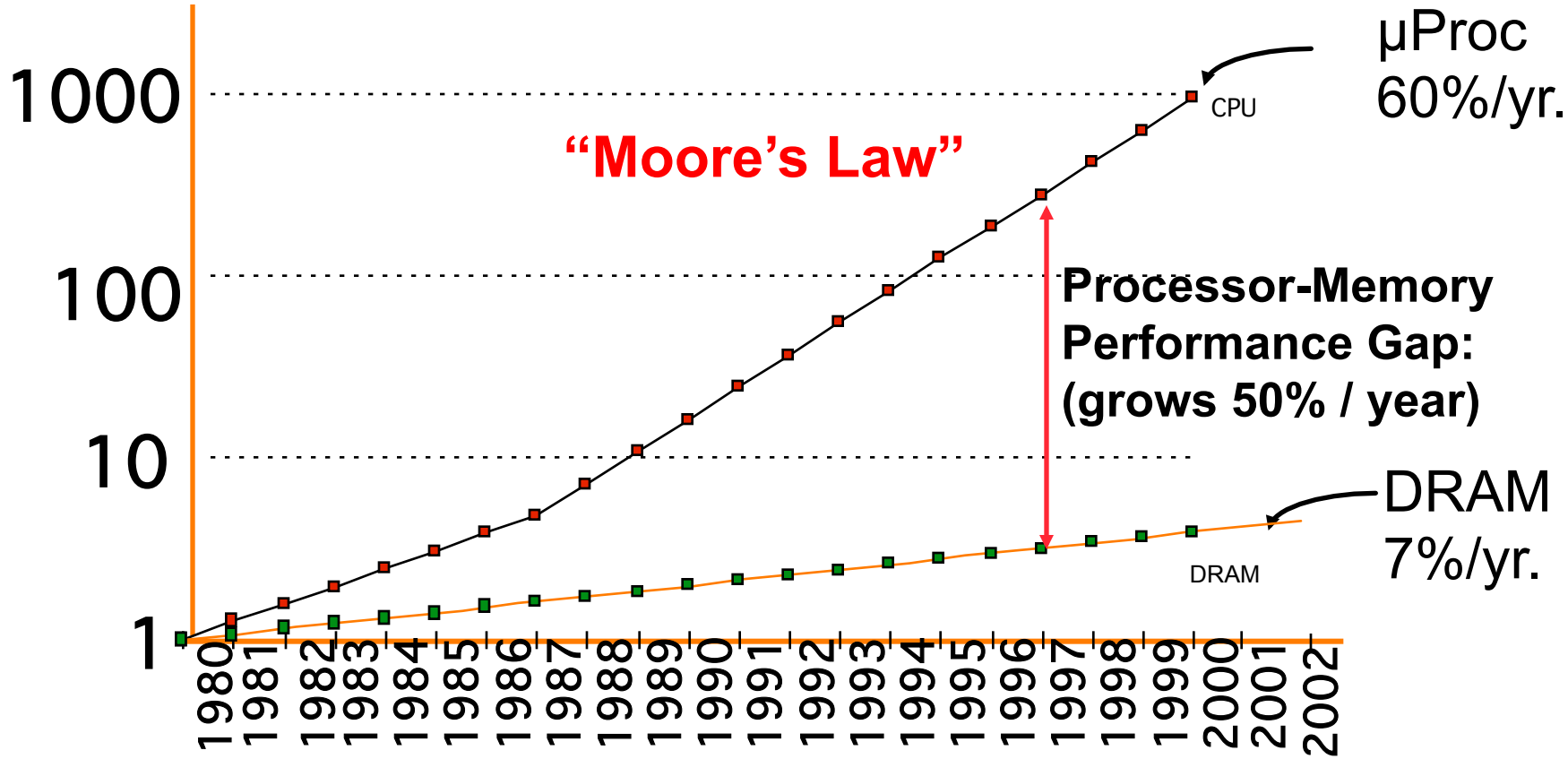
Message 1: Memory is Slow

- Inner core of processor, the CPU or the *μ*P, is getting faster at a steady rate
- Access to memory is also getting faster over time, but at a slower rate. Rate differential has existed for quite some time, with strange effect that fast processors rely on progressively slower memories
- Not uncommon on MP server that processor has to wait ~100 cycles before a memory access completes; that is one single memory access. On a Multi-Processor the bus protocol is more complex due to snooping, backing-off, arbitration, thus the number of cycles to complete a memory access can grow high
- IO compounds the problem of slow memory access

Message 1: Memory is Slow

- Discarding conventional memory altogether, relying only on cache-like memories, is NOT an option for 64-bit architectures, due to the price/size/cost/power if you pursue full memory population with 2^{64} bytes
- Another way of seeing this: Using solely reasonably-priced cache memories (say at < 10 times the cost of regular memory) is not feasible: the resulting physical address space would be too small, or the price too high
- Significant intellectual efforts in computer architecture focuses on reducing the performance impact of fast processors accessing slow, virtualized memories
- All else except IO, seems easy compared to this fundamental problem!
- IO is even slower by further orders of magnitude

Message 1: Memory is Slow



Source: David Patterson, UC Berkeley

Message 2: Events Tend to Cluster

- A strange thing happens during program execution: Seemingly unrelated events tend to cluster
- *memory accesses* tend to concentrate a majority of their referenced addresses onto a small domain of the total address space. Even if all of memory is accessed, during some periods of time such clustering is observed. Intuitively, one memory access seems independent of another, but they both happen to fall onto the same page (or working set of pages)
- We call this phenomenon *Locality*! Architects exploit locality to speed up memory access via *Caches* and increase the address range beyond physical memory via *Virtual Memory Management*. Distinguish *spacial* from *temporal* locality

Message 2: Events Tend to Cluster

- Similarly, hash functions tend to concentrate an unproportionally large number of keys onto a small number of table entries
- Incoming search key (say, a C++ program identifier) is mapped into an index, but the next, completely unrelated key, happens to map onto the same index. In an extreme case, this may render a hash lookup slower than a sequential, linear search
- Programmer must *watch out* for the phenomenon of clustering, as it is undesired in hashing!

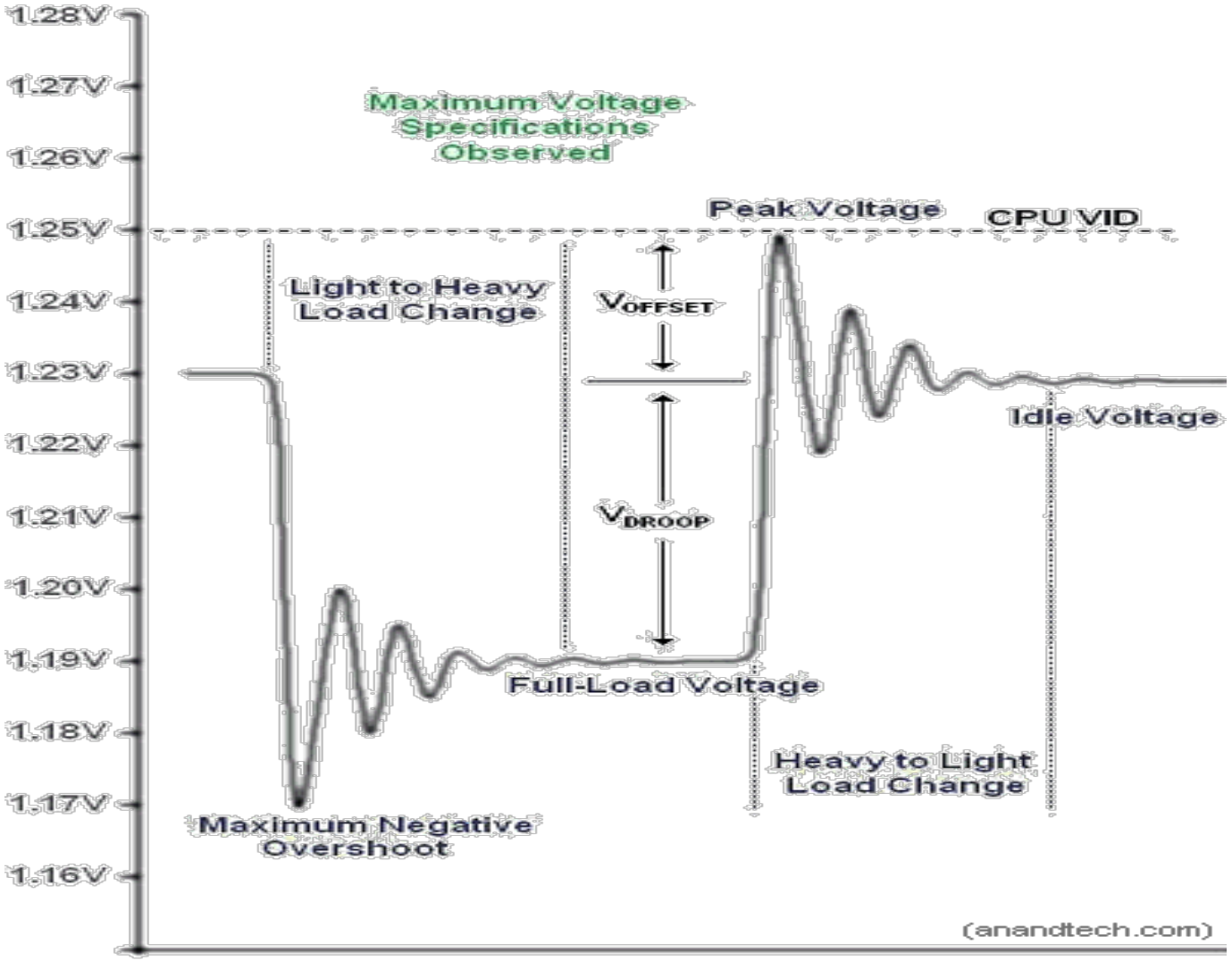
Message 2: Events Tend to Cluster

- Clustering happens in all diverse modules of the processor architecture. For example, when a data cache is used to speed-up memory accesses by having a copy of frequently used data in a faster memory unit, it happens that a small cache suffices to speed up execution
- Due to *Data Locality* (spatial and temporal). Data that have been accessed recently will again be accessed in the near future, or at least data that live close by will be accessed in the near future
- Thus they happen to reside in the same cache line. Architects do exploit this to speed up execution, while keeping the incremental cost for HW contained. Here clustering is a valuable phenomenon

Message 3: Heat is Bad

- Clocking a processor fast (e.g. > 3-5 GHz) can increase performance and thus generally "is good"
- Other performance parameters, such as memory access speed, peripheral access, etc. do not scale with the clock speed. Still, increasing the clock to a higher rate is desirable
- Comes at the cost of higher current, thus more heat generated in the identical physical geometry (the real-estate) of the silicon processor or also the chipset
- But the silicon part acts like a heat-conductor, conducting better, as it gets warmer (negative temperature coefficient resistor, or NTC). Since the power-supply is a constant-current source, a lower resistance causes lower voltage, shown as V_{Droop} in the next figure below

Message 3: Heat is Bad



Message 3: Heat is Bad

- This in turn means, voltage must be increased artificially, to sustain the clock rate, creating more heat, ultimately leading to self-destruction of the part
- Great efforts are being made to increase the clock speed, requiring more voltage, while at the same time reducing heat generation. Current technologies include sleep-states of the Silicon part (processor as well as chip-set), and *Turbo Boost* mode, to contain heat generation while boosting clock speed just at the right time
- Good that to date Silicon manufacturing technologies allow shrinking of transistors and thus of whole dies
- Else CPUs would become larger, more expensive, and above all: hotter

Message 4: Resource Replication

- Architects cannot increase clock speed beyond physical limitations
- One cannot decrease the die size beyond evolving technology
- Yet speed improvements are desired, and must be achieved
- This conflict can partly be overcome with replicated resources! But careful!
- Why careful? Resources could be used for better purpose!

Message 4: Resource Replication

- Key obstacle to parallel execution is data dependence in the SW under execution. A datum cannot be used, before it has been computed
- Compiler optimization technology calls this use-def dependence (short for use-before-definition), AKA true dependence, AKA data dependence
- Goal is to search for program portions that are independent of one another. This can be at multiple levels of focus

Message 4: Resource Replication

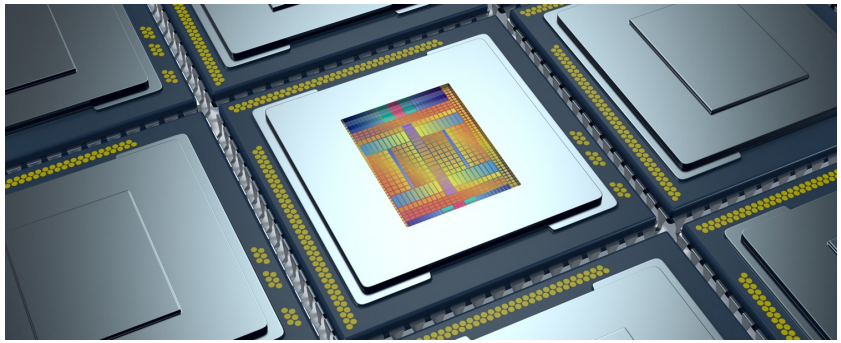
- At the very low level of registers, at the machine level –done by HW; see also score board
- At the low level of individual machine instructions –done by HW; see also superscalar architecture
- At the medium level of subexpressions in a program –done by compiler; see CSE
- At the higher level of several statements written in sequence in high-level language program –done by optimizing compiler or by programmer
- Or at the very high level of different applications, running on the same computer, but with independent data, separate computations, and independent results –done by the user running concurrent programs

Message 4: Resource Replication

- Whenever program portions are independent of one another, they can be computed at the same time: in parallel; but will they?
- Architects provide resources for this parallelism
- Compilers need to uncover opportunities for parallelism
- If two actions are independent of one another, they can be computed simultaneously
- Provided that HW resources exist, that the absence of dependence has been proven, that independent execution paths are scheduled on these replicated HW resources

Amdahl's Law

- Articulated by Gene Amdahl
- During 1967 AFIPS conference
- Stating that the maximum speedup of a program P is dominated by its sequential portion S
- I.e. if some part of program P can be perfectly accelerated due to numerous parallel processors, but some part S of P in inherently sequential, then the resulting performance is dominated by S
- See sample:



Amdahl's Law (From Wikipedia)

The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors are used

n = element of N, N number of processors

B = element of { 0, 1 }

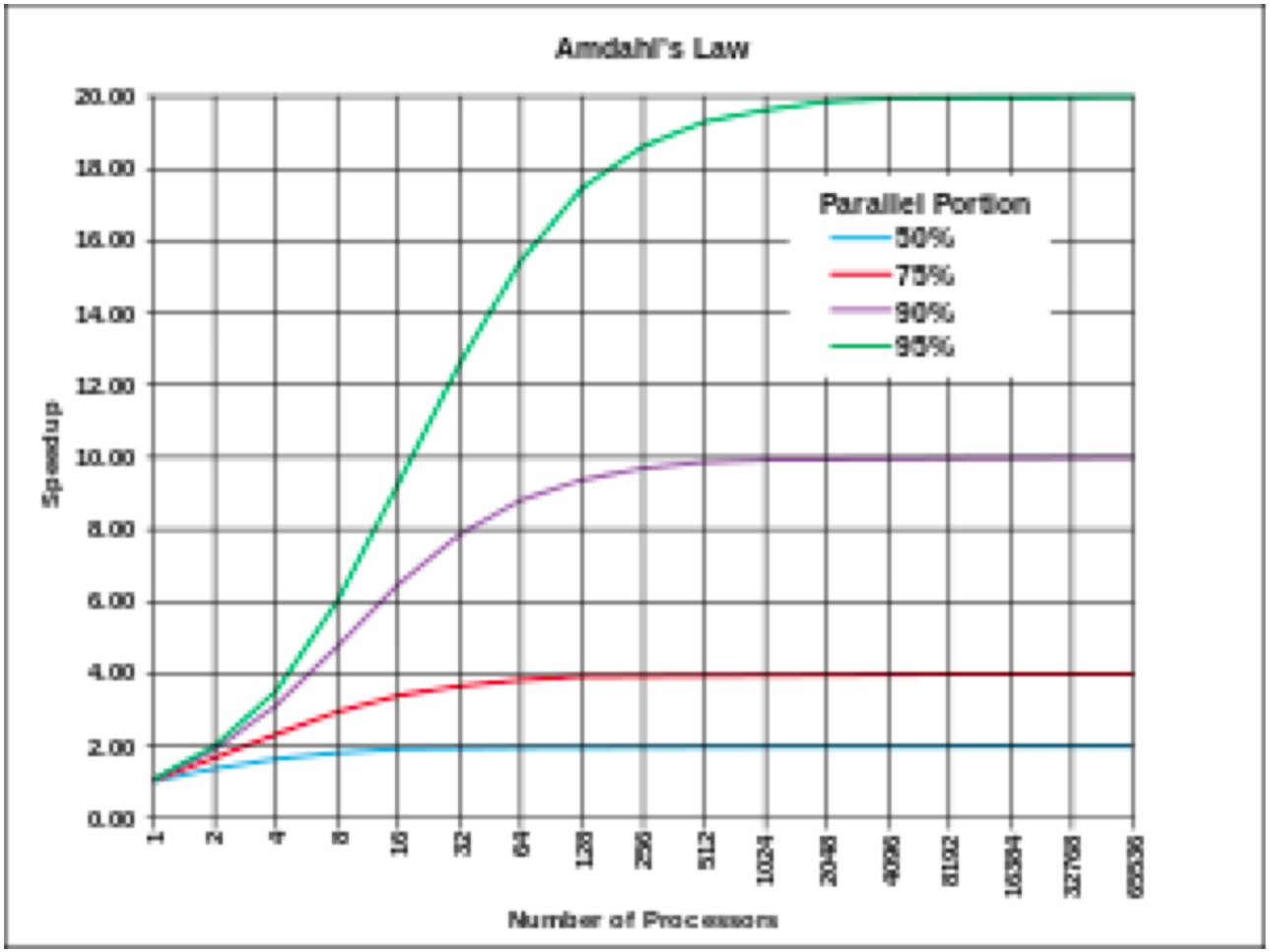
T(n) = time to execute with n processors

T(n) = T(1) (B + (1-B) / n)

S(n) = Speedup T(1) / T(n)

S(n) = 1 / (B + (1 – B) / n)

Amdahl's Law (From Wikipedia)



Uniprocessor (UP) Architectures

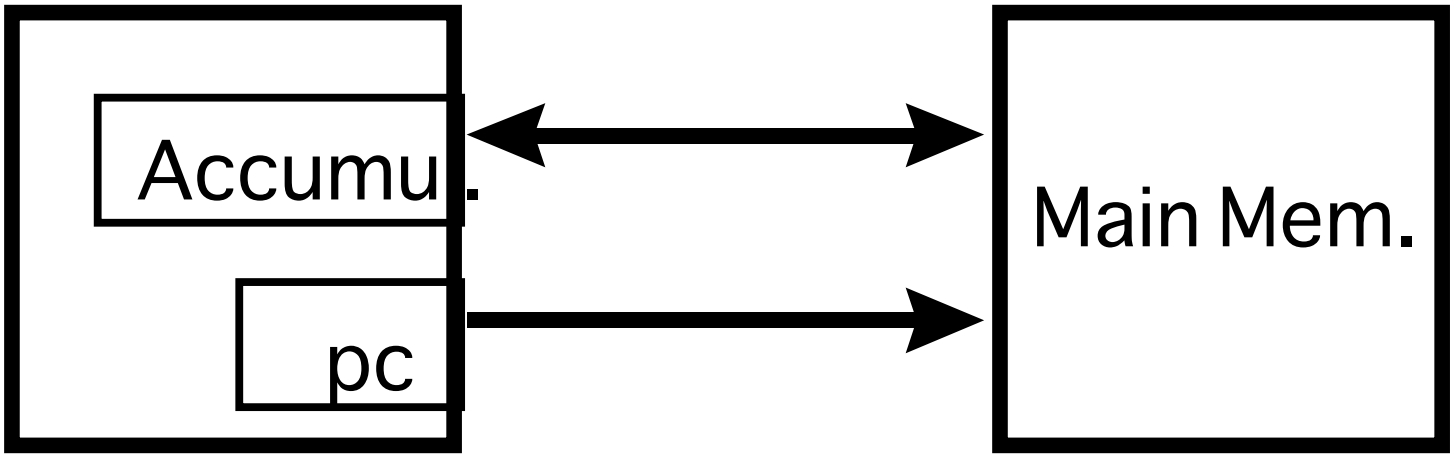
Ancient! Not used today for general computing:

- Single Accumulator (SAA) Architecture, e.g. Von Neumann's machine, in the 1940s
- Single register to hold operation results
- Conventionally called ***accumulator***
- Accumulator used as destination of arithmetic operations, and as (one) source
- Has central processing unit, memory unit, connecting memory bus
- pc points to next instruction (in memory) to be executed next
- Commercial sample: ENIAC

Architecture Evolution

Uniprocessor (UP) Architectures

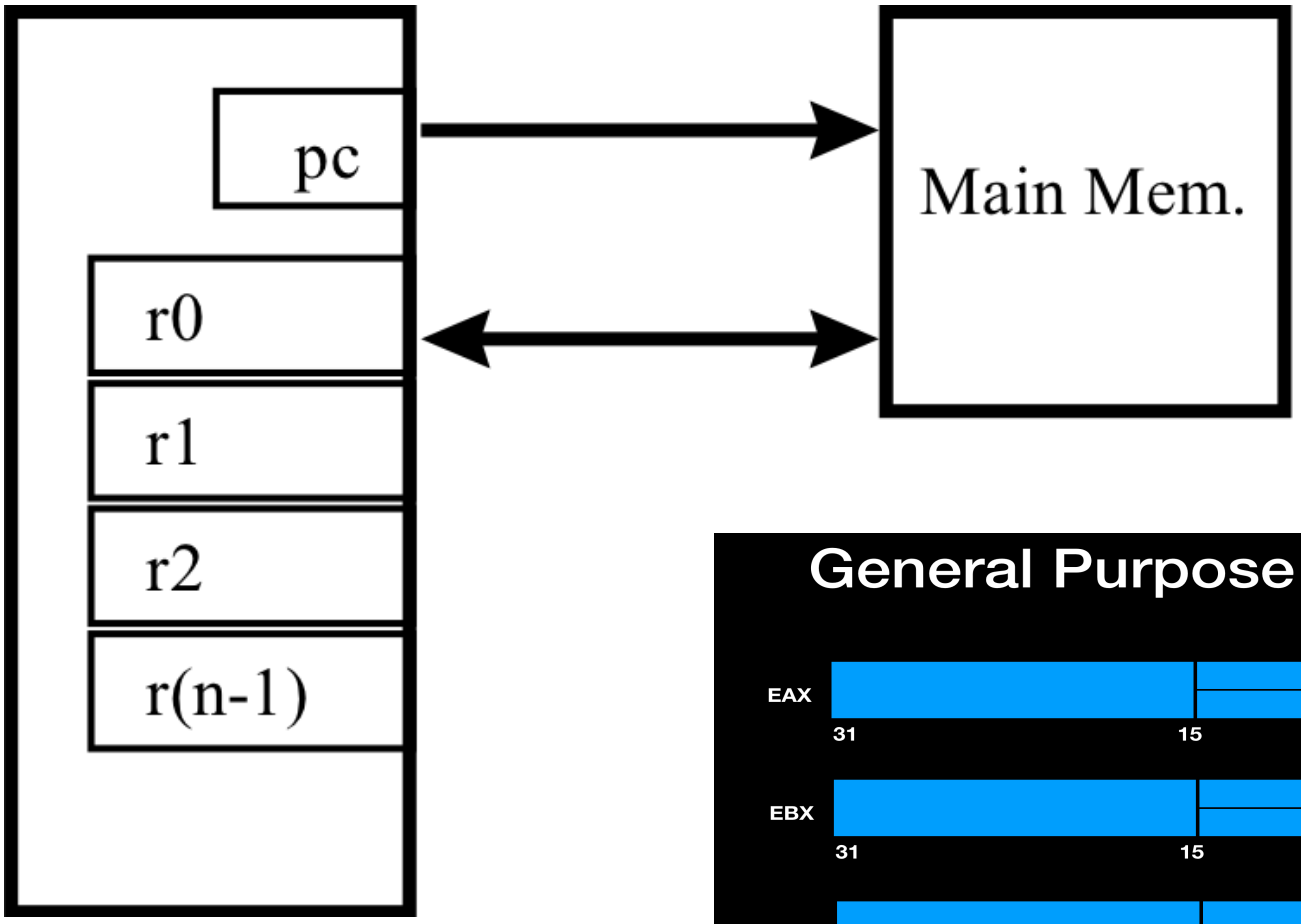
(See also page 16 above: Computer Modules)

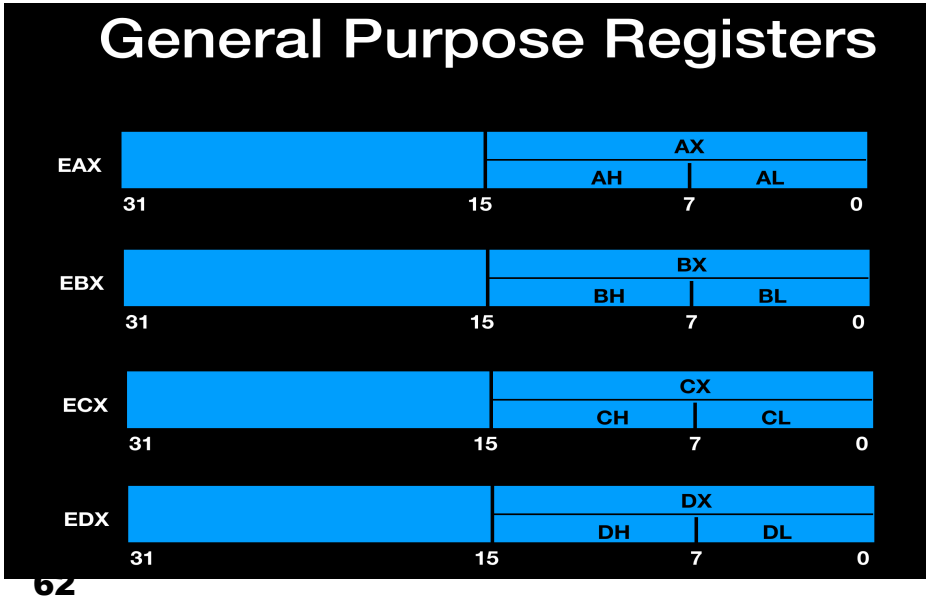


General-Purpose Register (GPR)

- Accumulates ALU results in n registers, n was typically 4, 8, 16, 64 (64 was long considered “large”)
- Allows register-to-register operations, fast!
- GPR is essentially a multi-register extension of SA architecture
- Two-address architecture specifies one source operand explicitly, another implicitly, plus one destination
- Three-address architecture specifies two source operands explicitly, plus an explicit destination
- Variations allow additional index registers, base registers, multiple index registers, etc.

General-Purpose Register (GPR)





Stack Machine Architecture (SMA)

- AKA zero-address architecture, since arithmetic operations require no explicit operand, hence no operand addresses; all are implied except for push and pop
- Students: What is equivalent of push/pop on GPR?
- Pure Stack Machine (SMA) has no registers
- Hence performance would be poor, as all operations involve memory!
- However, one can design an SMA that implements n top of stack elements as registers: Stack Cache
- Sample architectures: Burroughs B5000 family, and HP 3000 family

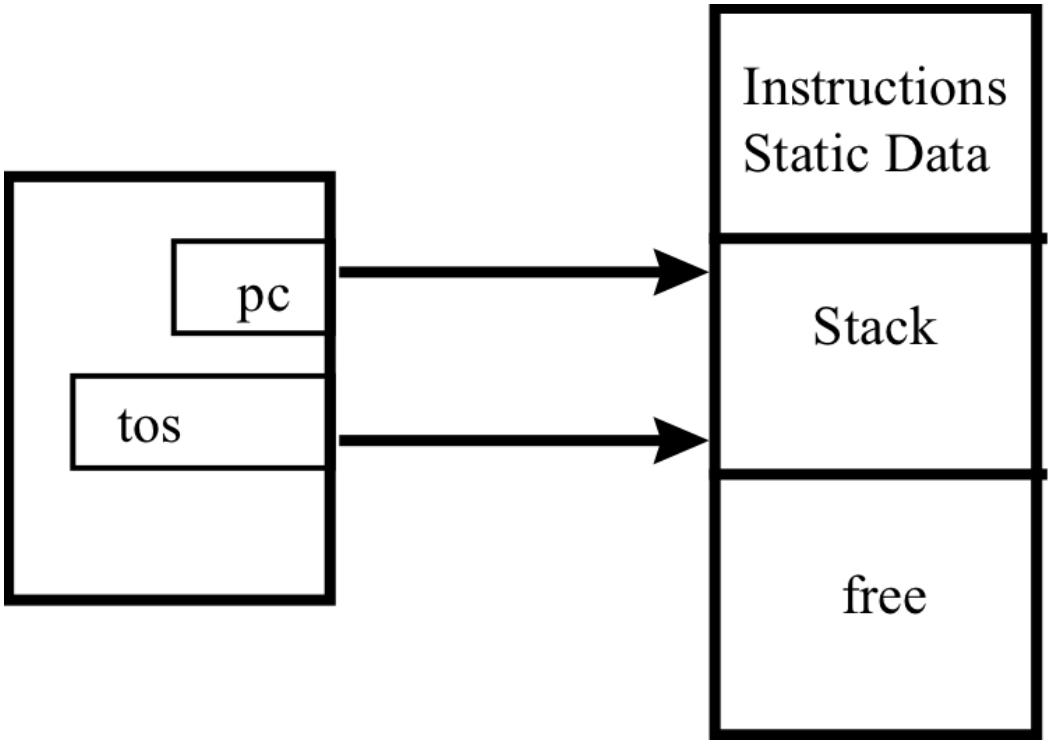
Stack Machine Architecture (SMA)

- Implement impure stack operations that bypass *tos* operand addressing
- Sample code sequence on SMA, first source, then SMA object code:

```
res := a * ( 145 + b ) -- operand sizes implied!

push      a     -- destination implied: stack
pushlit   145   -- also destination implied
push      b     -- ditto
add             -- 2 sources, and destination implied
mult            -- 2 sources, and destination implied
pop       res   -- source implied: stack
```

Stack Machine Architecture (SMA)



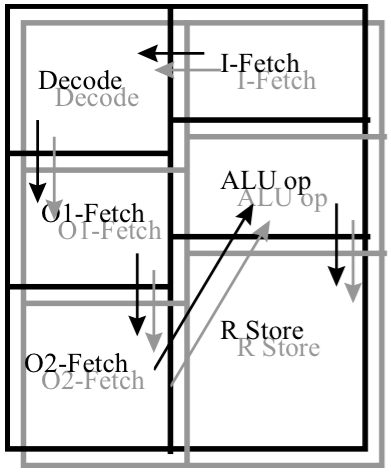
Pipelined Architecture (PA)

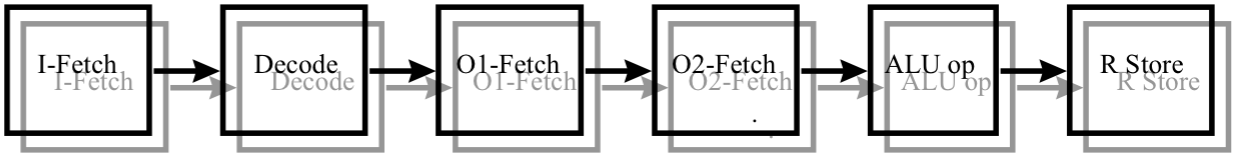
- Arithmetic Logic Unit, ALU, split into separate, sequentially connected units in PA
- Unit is referred to as a “stage”; more precisely the time at which the action is done is “the stage”
- Each of these stages/units can be initiated once per cycle
- Yet each subunit is implemented in HW just once
- Multiple subunits operate in parallel on different sub-ops, each executing a different stage; each stage is part instruction execution

Pipelined Architecture (PA)

- Non-unit time, differing # of cycles per operation cause different terminations
- Operations can abort in intermediate stage, if a later instruction changes the flow of control
- E.g. due to a branch, exception, return, conditional branch, call
- Operation must *stall* in case of operand dependence: stall, caused by interlock; AKA dependency of data or control

Pipelined Architecture (PA)





Pipelined Architecture (PA)

- Ideally each instruction can be partitioned into the same number of stages, i.e. sub-operations
- Operations to be pipelined can sometimes be evenly partitioned into equal-length sub-operations
- That equal-length time quantum might as well be a single sub-clock
- In practice hard for architect to achieve
- Compare for example times for integer add and floating point divide!
- Vastly different time needs due to different complexities of tasks!

Pipelined Architecture (PA)

- Ideally all operations have independent operands
- i.e. one operand being computed is not needed as source of the next few operations
- if they were needed –and often they are—then this would cause ***dependence,*** which causes a ***stall***
- read after write (RAW)
- write after read (WAR)
- write after write –with use in between (WAW)
- Also, ideally, all instructions just happen to arranged sequentially one after another
- In reality, there are branches, calls, returns etc.

Pipelined Architecture (PA)

Idealized Pipeline Resource Diagram:

	1	2	3	4	5	6	7	8	9	10	11	12	13	time
I8								**if**	**de**	**op1**	**op2**	**exec**	**wb**	
I7							**if**	**de**	**op1**	**op2**	**exec**	**wb**		
I6						**if**	**de**	**op1**	**op2**	**exec**	**wb**			
I5					**if**	**de**	**op1**	**op2**	**exec**	**wb**				
I4				**if**	**de**	**op1**	**op2**	**exec**	**wb**					
I3			**if**	**de**	**op1**	**op2**	**exec**	**wb**						
I2		**if**	**de**	**op1**	**op2**	**exec**	**wb**							
I1	**if**	**de**	**op1**	**op2**	**exec**	**wb**								
						↑ I1	**↑ I2**	**↑ I3**	**↑ I4**	**↑ I5**	**↑ I6**	**↑ I7**	**↑ I8**	

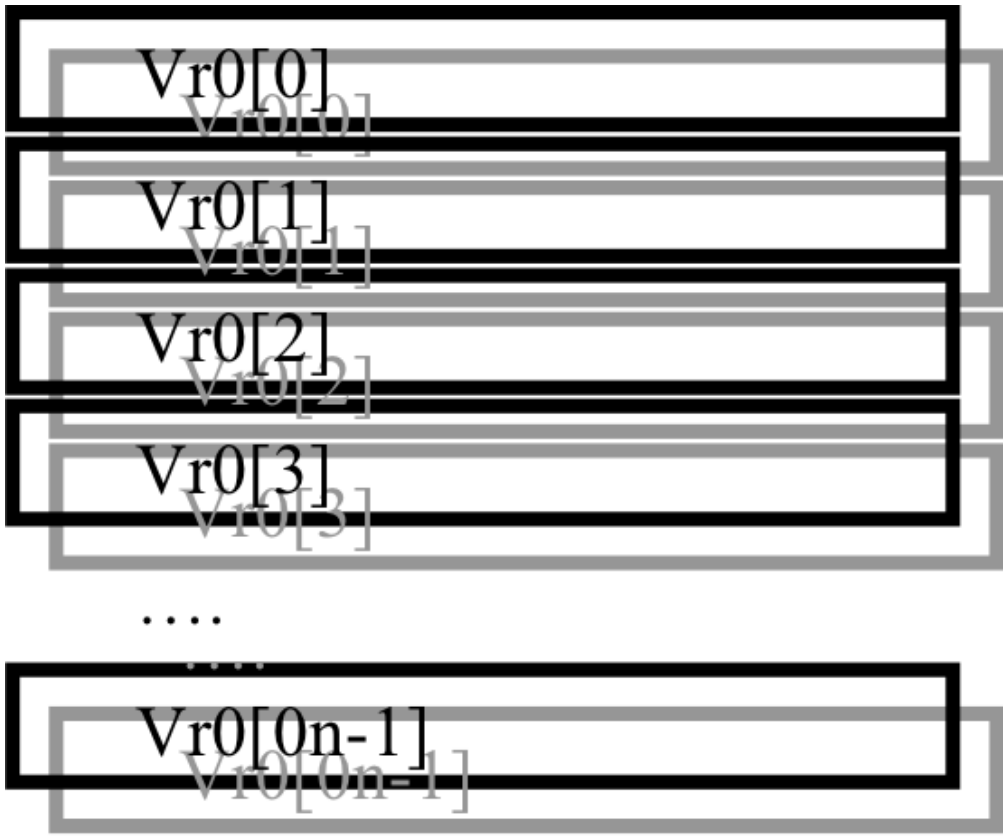
- if: fetch an instruction
- de: decode the instruction
- op1: fetch or generate the first operand; if any
- op2: fetch or generate the second operand; if any
- exec: execute that stage of the overall operation
- wb: write result back to destination, if any
  e.g. noop has no destination; halt has no destination, etc.

Vector Architecture (VA)

- Register implemented as HW array of identical registers, named vri
- VA may also have scalar registers, named r0, r1, etc.
- Scalar register can also be the first (index 0) of the vector registers
- Vector registers can load/store block of contiguous data
- Still in sequence, but overlapped; number of steps to complete load/store of a vector also depends on width of bus
- Vector registers can perform multiple operations of the same kind on whole contiguous blocks of operands

Vector Architecture (VA)

Still in sequence, but overlapped, and all n operands are readily available; vector register Vr[] shown here



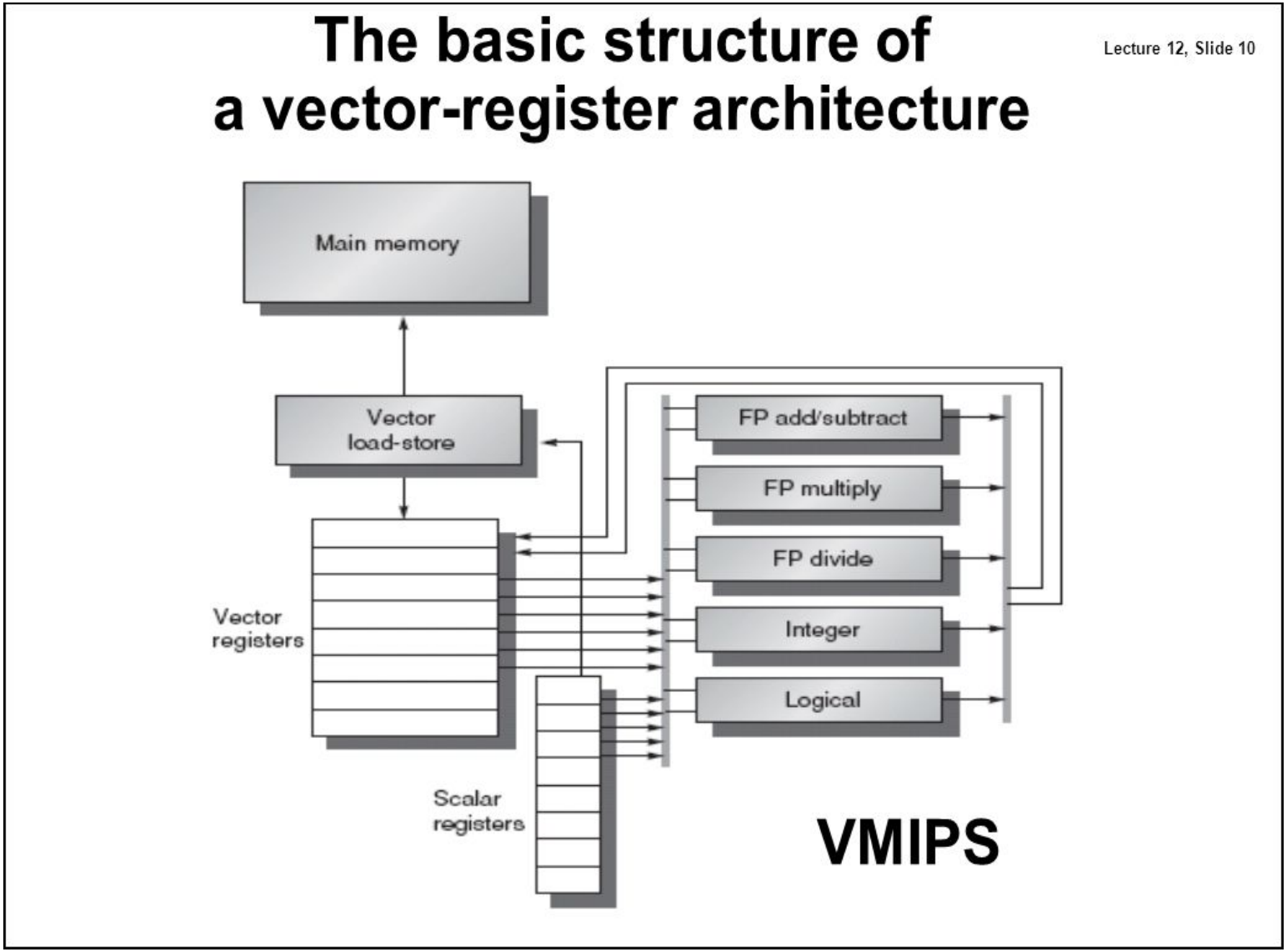
Vector Architecture (VA)

- Otherwise operation like GPR architecture
- Sample vector operations, assume 64-unit vector ops:

```
ldv vr1, mem_i                -- loads 64 memory locs from [mem+i=0..63]
stv vr2, mem_j                -- stores vr2 in 64 contiguous locs
vadd    vr1, vr2, vr3         -- register-register vector add
cvaddf r0, vr1, vr2, vr3      -- has conditional meaning:
-- sequential equivalent:
for i = 0 to 63 do
   if bit i in r0 is 1 then
    vr1[i] = vr2[i] + vr3[i]
   else
    -- do not move corresponding bits
   end if
end for
-- parallel syntax equivalent:
forall i = 0 to 63 doparallel
   if bit i in r0 is 1 then
    vr1[i] = vr2[i] + vr3[i]
   end if
end parallel for
```

Vector Architecture (VA)

Instead of one, VA has a list of multiple registers each



Multiprocessor (MP) Architectures

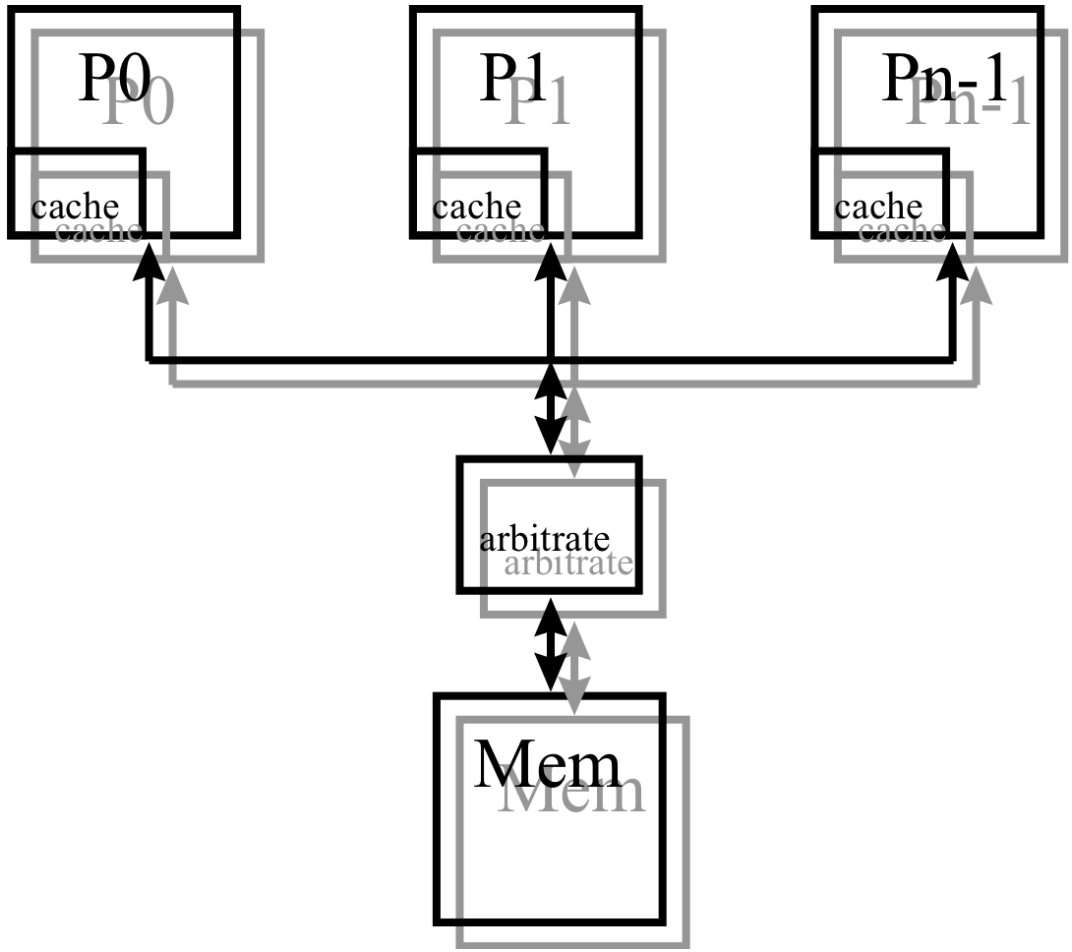
- ***Shared Memory Architecture (SMA)***
- **Equal access to memory for all n processors, p0 to pn-1**
- **Only one will succeed in accessing shared memory, if there are multiple, simultaneous accesses**
- **Simultaneous access must be deterministic; needed a policy or an arbiter that is deterministic**
- **Von Neumann bottleneck even tighter than for conventional UP system**
- **Typically there are twice as many loads as stores**

Multiprocessor (MP) Architectures

- Generally, some processors are idle due to memory or other conflict
- Typical number of processors n=4, but n=8 and greater possible, with large 2nd level cache, even larger 3rd level
- Early MP architectures had only limited commercial success and acceptance, due to programming burden, frequently burden on programmer
- Morphing in 2000s into multi-core and hyper-threaded architectures, where programming burden is on multi-threading OS

Multiprocessor (MP) Architectures

Yes, 3 CPUs --to make point ☺ -- with Shared Memory



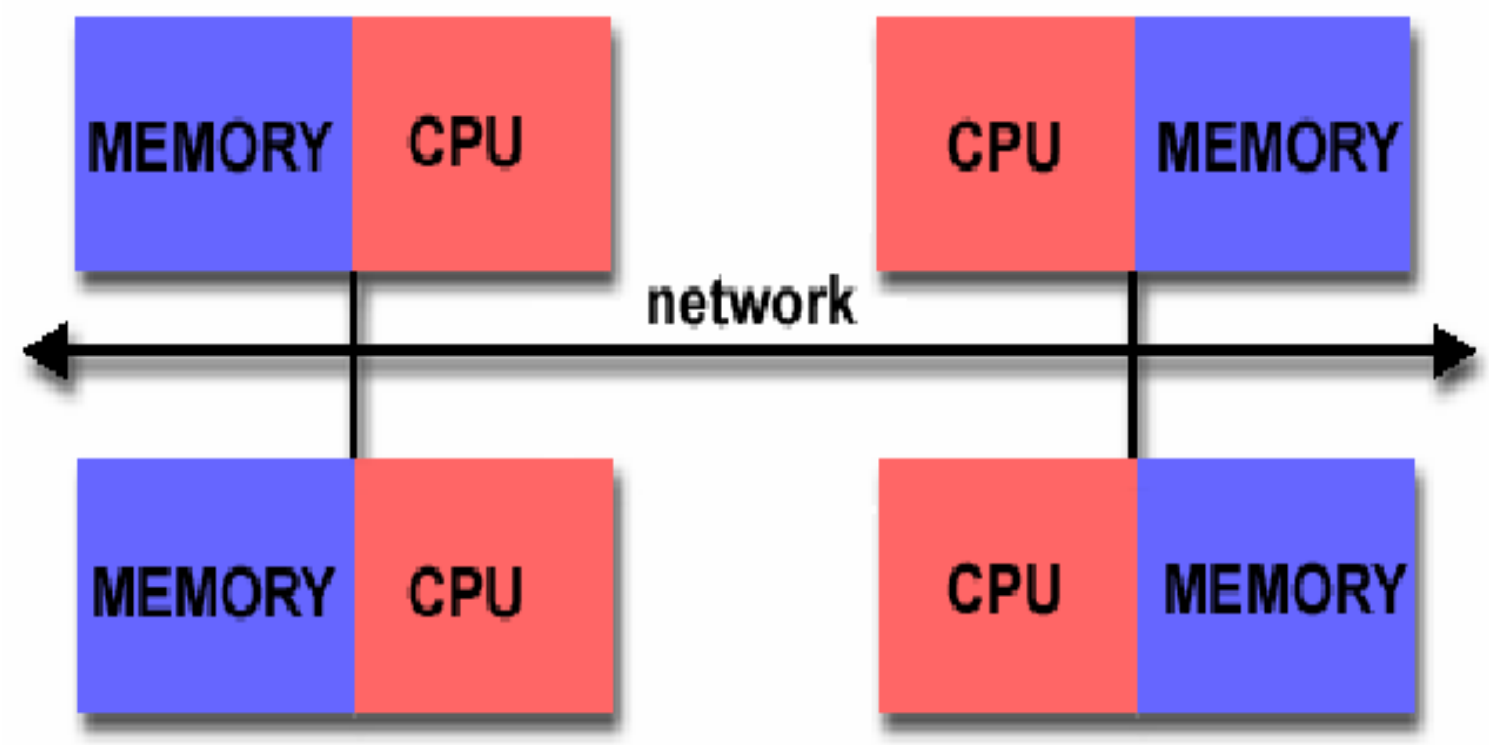
Distributed Memory Architecture

- Processors have private, AKA local memories
- Yet programmer has to see single, logical memory space, regardless of local distribution
- Hence each processor p_i always has access to its own memory Mem_i
- And collection of all memories Mem_i i= 0..n-1 is program's *logical data space*
- Thus, processors must access others' memories
- Done via *Message Passing* or *Virtual Shared Memory*
- Messages must be routed, route be determined
- Route may require multiple, intermediate nodes

Distributed Memory Architecture

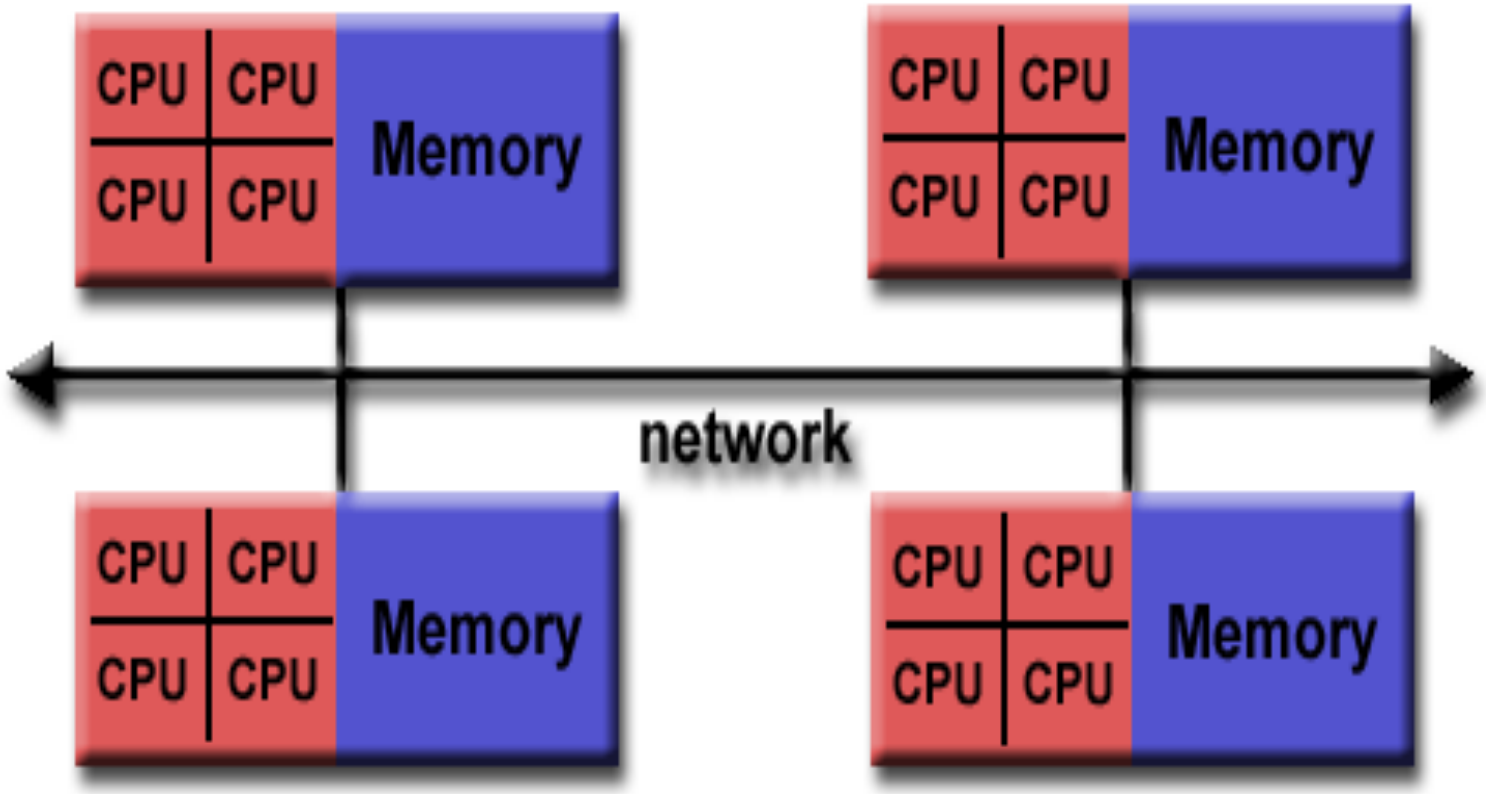
Four CPUs each with own (distributed) memory

Network or Bus connects all, though slowly: multi-cycle



Distributed Memory MP Architecture

Four 4-way MP CPUs with distributed memory



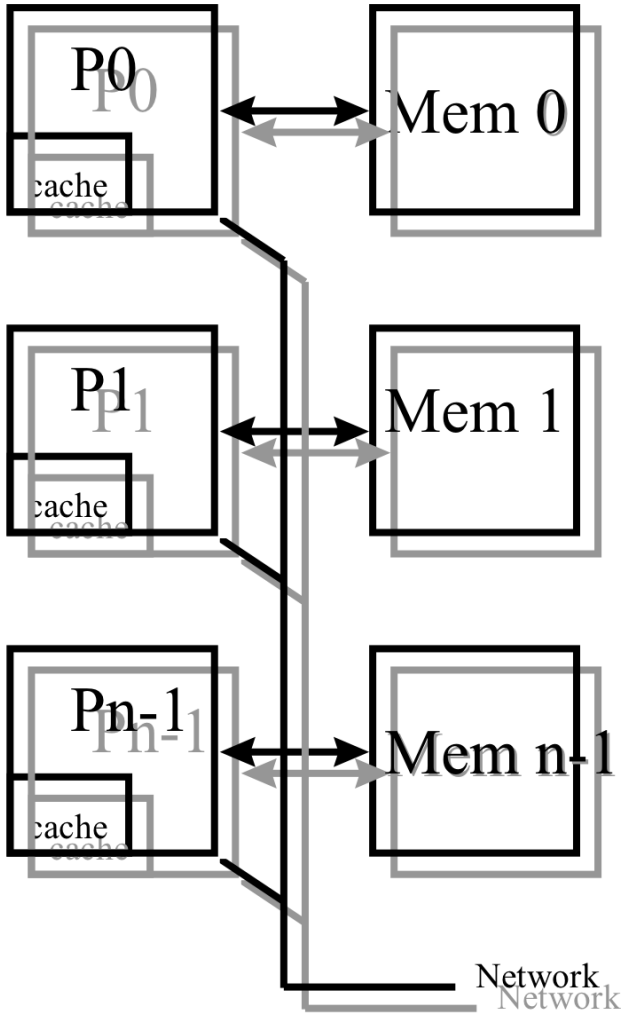
Distributed Memory Architecture

- Blocking when: message expected but hasn't arrived yet
- Blocking when: message to be sent, but destination cannot receive
- Growing message buffer size increases illusion of asynchronicity of sending and receiving operations
- Key parameter: time for 1 hop and package overhead to send empty message
- Message may also be delayed because of network congestion!

Distributed Memory Architecture

Another view: 3-way MP processor, each with its own, i.e. distributed memory.

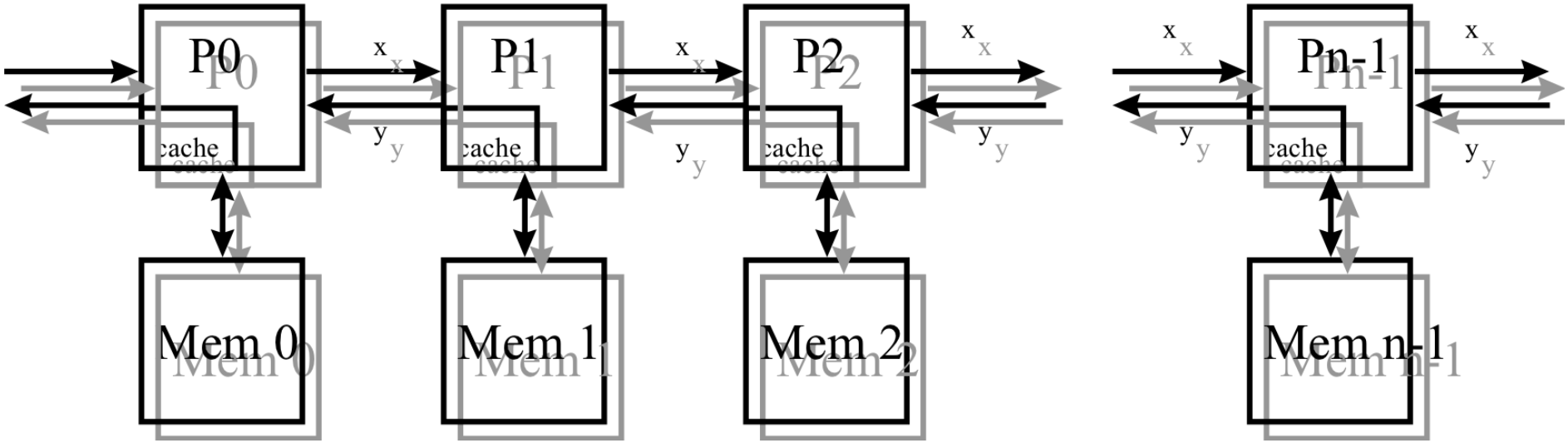
Can "share" data via a network that renders all memory visible to all CPUs, but at cost of multiple cycles for sending + receiving!



Systolic Array (SA) Architecture

- Very few designed: CMU and Intel for (then) ARPA
- Each processor has private memory
- Network is pre-defined by the *Systolic Pathway (SP)*
- Each node is pre-connected via *SP* to some subset of other processors
- Node connectivity: determined by implemented/ selected network topology
- Systolic pathway is high-performance network; sending and receiving may be synchronized (blocking) or asynchronous (data received are buffered)
- Typical network topologies: line, ring, torus, hex grid, mesh, etc.

1-D Systolic Array (SA) Architecture



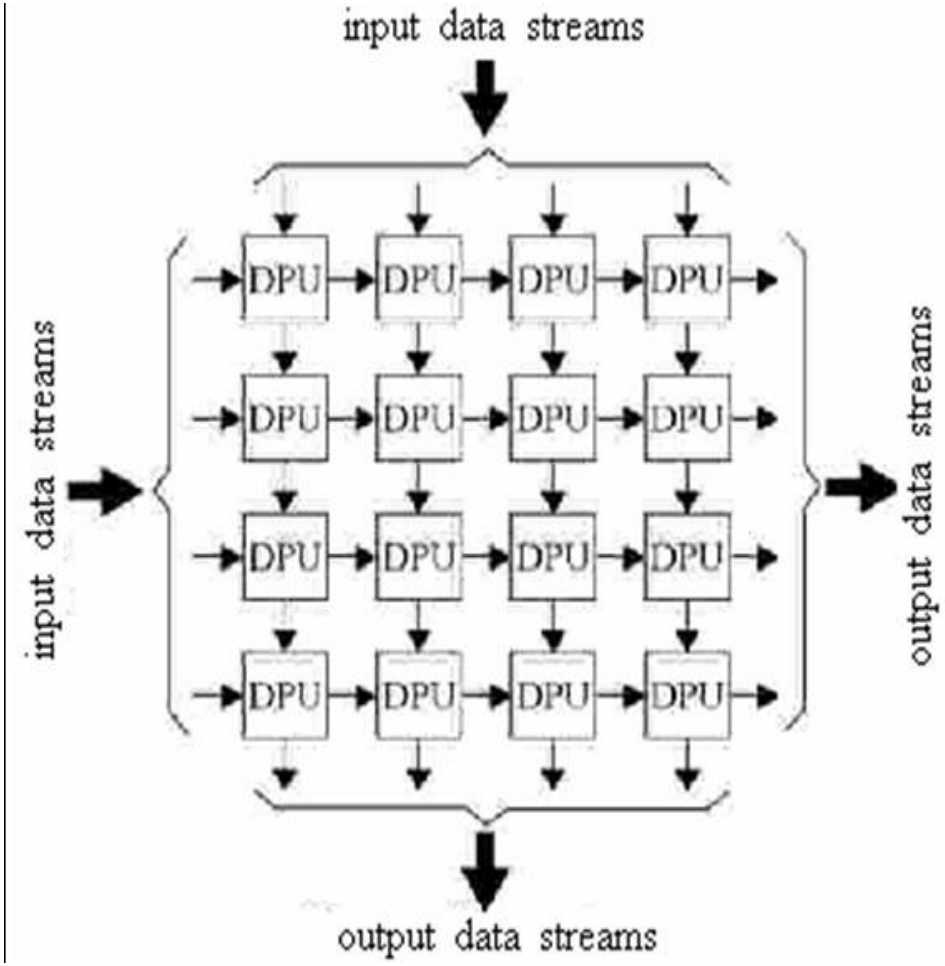
Systolic Array (SA) Architecture

- Note that each pathway, x or y, may be bi-directional
- May have any number of pathways, nothing magic about 2, x and y
- Possible to have I/O capability with each node
- Typical application: large polynomials of the form:

$$y = k_0 + k_1 * x^1 + k_2 * x^2 \; .. \; + k_{n-1} * x^{n-1} = \Sigma \; k_i * x^i$$

Next example shows a torus without displaying the wrap-around pathways across both dimensions

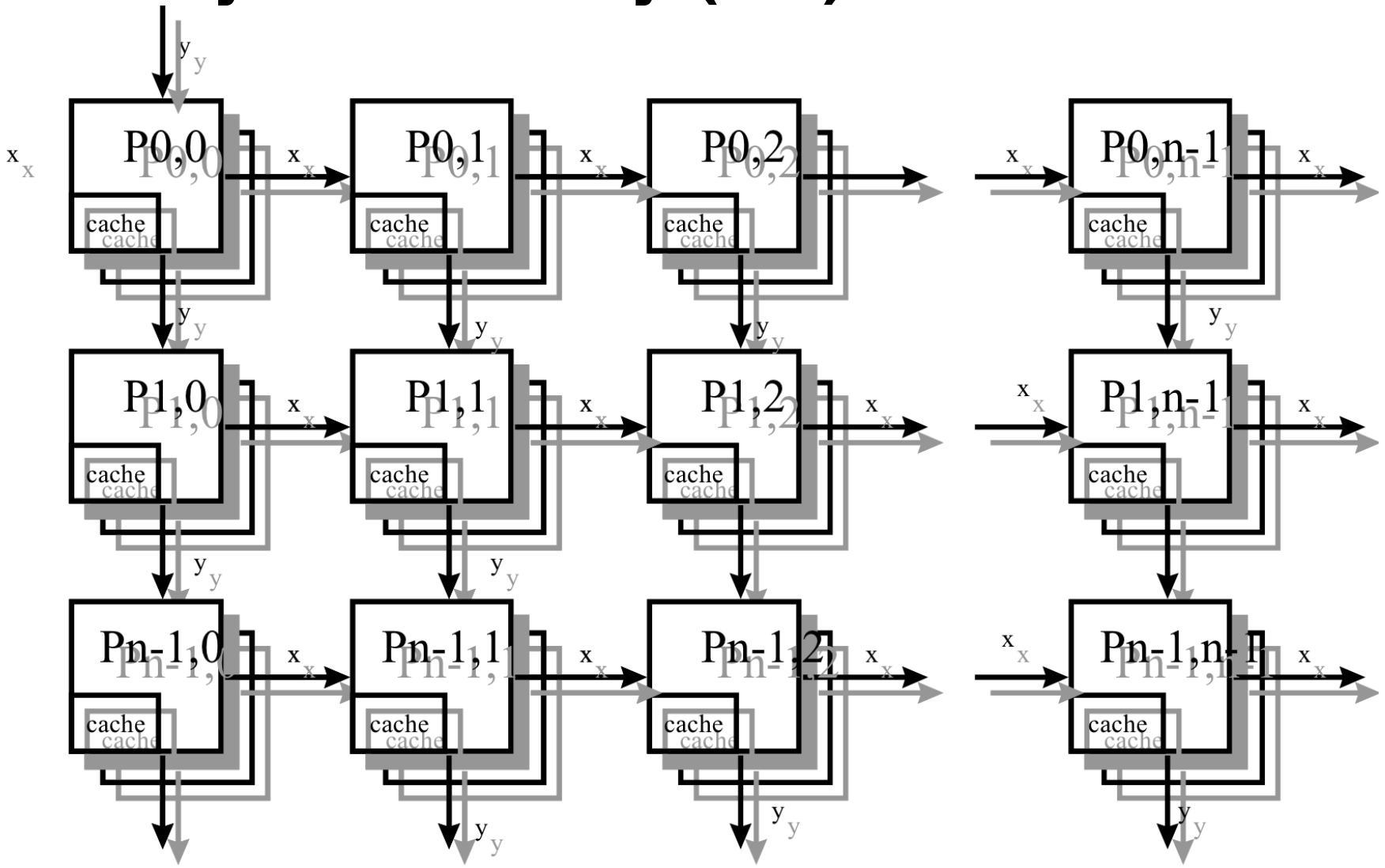
2-D Systolic Array (SA) Architecture



Systolic Array (SA) Architecture

- Sample below is a ring; note that wrap-around along x and y direction is not explicitly shown
- Processor can write to x or y gate; sends word off on x or y *SP*
- Processor can read from x or y gate; consumes word from x or y *SP*
- Buffered *SA* can write to gate, even if receiver cannot read
- Reading from gate when no message available blocks
- Automatic code generation for non-buffered *SA* hard, compiler must keep track of interprocessor synchronization
- Can view *SP* as an extension of memory with infinite capacity, but with sequential access

2-D Systolic Array (SA) Architecture



Hybrid Architectures

- ***Superscalar (SSA) Architecture***
- **Replicates (duplicates) some operations in HW**
- **Seems like scalar architecture w.r.t. object code**
- **Is parallel architecture, as it has multiple copies of some hardware units**
- **Is not MP architecture: the multiple units do not have concurrent, independent memory access**
- **Has multiple ALUs, possibly multiple FP add (FPA) units, FP multiply (FPM) units, and/or integer units**
- **Arithmetic operations simultaneous with load and store operations; note data dependence!**

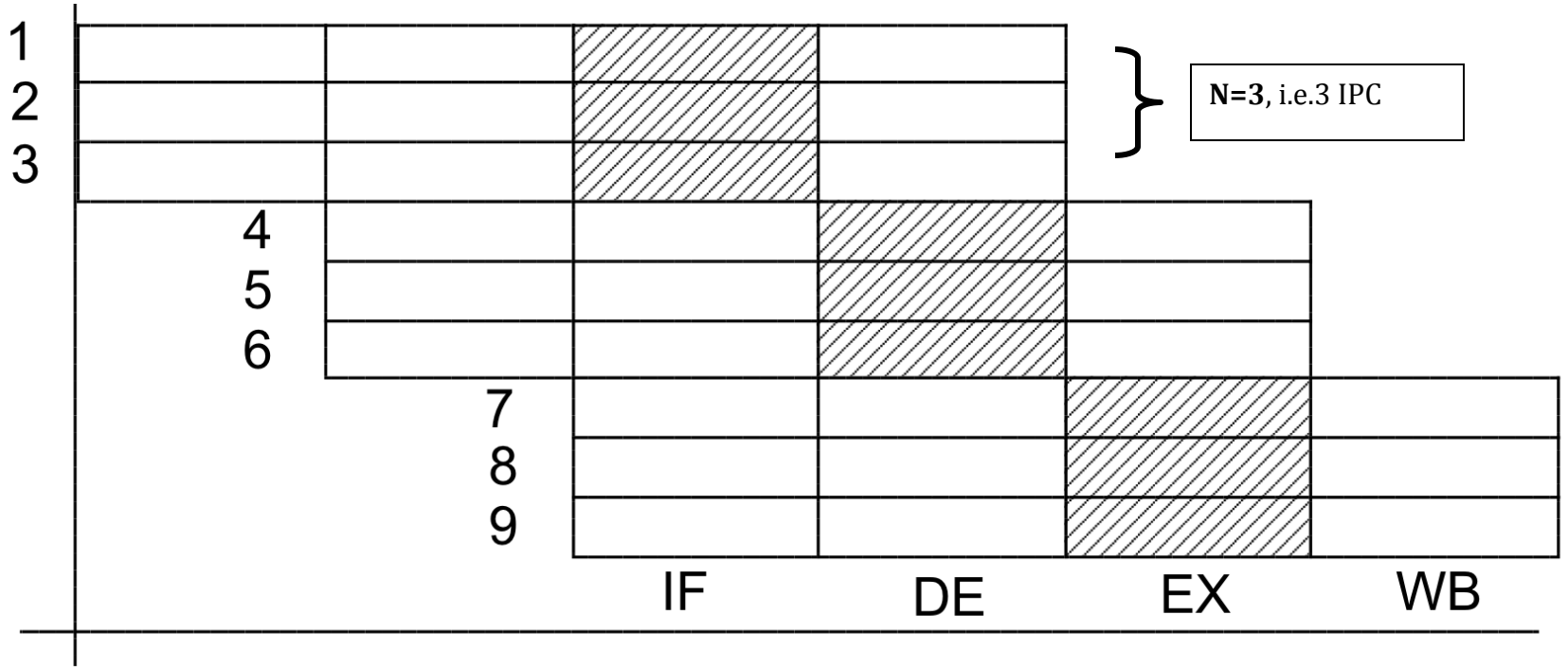
Hybrid Architectures

- Instruction fetch in superscalar architecture is speculative, since number of parallel operations unknown; rule: fetch too much! But fetch no more than longest possible superscalar pattern
- Code sequence looks like sequence of instructions for scalar processor
- Example: 80486® code executed on Pentium® processors
- More famous and successful example: 80860 processor; see below
- Object code can be custom-tailored by compiler; i.e. compiler can have superscalar target processor in mind, bias code emission, knowing that some code sequences are better suited for superscalar execution

Hybrid Architectures

- Fetch enough instruction bytes on superscalar target to support widest (most parallel) possible object sequence
- Decoding is bottle-neck for CISC, easier for RISC ⇒ 32-bit units, or 64-bit units
- Sample of superscalar: i80860 has separate FPA, FPM, 2 integer ops, load, store with pre-post address-increment and decrement
- Superscalar, pipelined architecture with maximum of 3 instructions per cycle; here the pipelined stages are: IF, DE, EX, and WB

Hybrid Architectures



VLIW Architecture (VLIW)

- Very Long Instruction Word, typically 128 bits or more
- Object code is no longer purely scalar, but explicitly parallel, though parallelism cannot always be exploited
- Just like limitation in superscalar: This is not a general MP architecture: The subinstructions do not have concurrent memory access; dependences have to be resolved before code emission
- But VLIW opcodes are designed to support some parallel execution
- Compiler/programmer explicitly packs parallelizable operations into VLIW instruction

VLIW Architecture (VLIW)

- Just like *horizontal microcode compaction*
- Other opcodes are still scalar, can coexist with VLIW instructions
- Partial parallel, even scalar, operations possible by placing no-ops into some of the VLIW fields
- Sample: Compute instruction of CMU warp® and Intel® iWarp®
- Could be 1-bit (or few-bit) opcode for compute instruction; plus sub-opcodes for subinstructions
- Data dependence example: Result of FPA cannot be used as operand for FPM in the same VLIW instruction

VLIW Architecture (VLIW)

- Result of int1 cannot be used as operand for int2, etc.
- Thus, need to *software-pipeline*
- Below: this is one VLIW instruction

VLIW opcode	FPA sub-op	FPM sub-op	int1 sub-op	int2 sub-op	load1 op	load2 op	store op

EPIC Architecture (EA)

- Groups instructions into bundles
- Straighten out branches by associating predicate with instructions
- Execute instructions in parallel, say the *else* clause and the *then* clause of an If Statement
- Decide at run time which of the predicates is true, and execute just that path of multiple choices
- Use speculation to straighten branch tree
- Use large, rotating register file
- Has many registers, not just 64 GPRs

Summary

- Broad classification by Flynn into SISD, SIMD, MISD, and MIMP
- ISA is boundary between running SW and hosting HW
- At times a thin layer of firmware can replace or simulate small HW portions
- Vector machine architecture executes the same instruction on array (the vector) or data
- Pipelined machine breaks instructions into sub operations and executes one sub operation of multiple sequential instructions in one step

References

1. http://www.csupomona.edu/~hnriley/www/VonN.html
2. http://cva.stanford.edu/classes/ee482s/scribed/lect11.pdf
3. VLIW Architecture:http://www.nxp.com/acrobat_download2/other/vliw-wp.pdf
4. ACM reference to Multiflow computer architecture: http://dl.acm.org/citation.cfm?id=110622&coll=portal&dl=ACM