Six, no! Eight, no! Eleven Memories for Computer Architecture

James C. Brakefield © 2003

Background:

- Taxonomy of computers:

Herb Mayer, Portland State University:
http://www.cs.pdx.edu/~herb/cs201s03/taxonomy.doc

Flynn MJ, Very High-Speed Computing Systems, Proceeding of the IEEE, 54(12), December 1966, p1901-1909 (SISD, SIMD, MISD, MIMD)

- Quantitative analysis of computer performance:

David A. Patterson & John L. Hennessy: Computer Architecture: A Quantitative Approach

- Case studies of computer architecture:

Gerrit Blaauw & Frederick Brooks Jr.: Computer Architecture: Concepts and Evolution
C. Gordon Bell & Allen Newell: Computer Structures: Readings and Examples

- Following presentation is a mixture of conceptual and real-world issues.

A Memory Based Approach to Computer Architecture

- Computer Architecture is usually taught by case studies

- Another approach is the analysis of performance.

- Both of these lack a unified sense of "style".
  Current computer architecture is extremely pragmatic.

- Architecture in buildings is a compromise between
  Engineering feasibility and cost versus
  Human usefulness and aesthetics

- Computer memory is generally overlooked as architecture
  Architectural elements are present
  Wonderful history of different technologies

Interpretation as the Overall Pattern

- Architecture Perspective used here:
  Computer design is the building of interpreters for the ISA (Instruction Set).

- Hardware
  Physical representation

- Emulation
  Software representation

- Micro-code
  Allows trading speed and cost

  Intel TCP/IP offload engine: 320 words, 80 bits, 10 Ghz
  (Microprocessor Report, 3/24/03, www.MPRonline.com)

Technology

- Past (e.g. 20+ years ago):
  - Circuit boards, Racks and Back-plane wiring
  - CPU Technology
    - Levers & gears; Relays; Vacuum tubes; Transistors; TTL…
  - Memory Technology
    - Wheels; Drums; Delay lines; Storage tube; Core memory; RAM…
  - ROM Technology
    - Plug board, Transformer, Capacitor matrix, Diode matrix, Silicon array
- Present:
  - ASIC & PLD: system on a chip (SOC)
  - CPU Technology
    - Silicon
  - Memory Technology
    - Silicon, no preferred size
    - For ROM, a choice of mapping into memory array or as logic
- Future (speculative):
  - Nano-computers
  - Protein computers

Virtualization:

- The software model of computer memory,
  As presented by the various programming languages and tools,
  Shapes the perception of computers and computer programming.

- And from the computer design point-of-view, the targeted market.

- The models have evolved over the years, generally in the direction of increased sophistication.

- Thus it is that the software model may have greater complexity than the hardware implementation, e.g., multiple "virtual" memory types may be folded into a single physical memory type.

The Thesis:

- Six necessary and "logical" uses for memory.
- Memory uses are not equally important or demanding.
- All computers contain some amount of each and in various ways.
- Study of computer architecture from the standpoint of memory is insightful.

- Development of CPLD and FPGA chips provides a rational for:
  Two additional memory spaces providing the
  Instruction set implementation

- Configuration bits in a CPLD or FPGA numbered and addressable

Abstract:

The first six memory mappings are architectural, they are distinct on a logical basis, but can be folded into one or more physical renderings in various ways for cost and performance reasons.

The two "implementation" memory regions of logic and connectivity map the processor (e.g. the instruction set) into some number of bits. Thus instruction set complexity and implementation structure can be mapped into memory and thereby quantified.

For completeness, I/O space and register space are easily numbered and made part of the overall "architectural memory space". Finally, FIFOs (e.g. message buffers) are ubiquitous in multi-processor systems.

The Six Architectural Memory Regions Are:

Program memory	Where the programmer places his code
Data memory	What the code operates on
Micro-code memory	"Subroutines" beneath the assembler level
Data stack	Where expression temporaries go
	>Where subroutine parameters & results go
Return address stack	Where return addresses go
Op-code lookup table	Maps op-codes into micro-code addresses

The Two "Implementation" Memory Regions Are:

- Logic as ROM

  ASIC – number of bits to enumerate gate library
  1, 2, 3… input gates (Not, Nand, Nor, Xor, Muxes)

  FPGA – number of bits in the lookup tables (LUTs) on a logical basis
  1, 2, 3, 4… input LUTs

- Connectivity as ROM The logic connectivity mapped into memory

  ASIC:
  Normalized wire length ??

  FPGA:
  Routing switches

  Emulation:
  Binary enumeration
  (Compression techniques can reduce bit count further)

The Three Ancillary Memory Regions Are:

• I/O space	Early I/O via special instructions, modern I/O via addressable regs
• Register space	For those that wish to number everything
• FIFOs/Buffers	For multiprocessor and/or multi-tasking systems

Future Memory (speculative):

• Manufacturing complexity	The cost of complexity mapped to bits
• Pattern Matching via Proteins	Connectivity via diffusion
Structural pattern from DNA bits	Protein folding problem
Control structure	Turning proteins/DNA on and off

#1 Program Memory Rationalization

- The program is an emulation device allowing the computer as a finite state machine to emulate the finite state machine embedded in the program.

- Another way to look at emulation: The programming language is a register transfer language suited for piecemeal state changes.

#2 Data Memory Rationalization

- Data can be considered as an abstraction or generalization if input and output.

- E.g. in the sense of the embodied robot:
  Data is internal state,
  I/O is external state.
  Rodney Brooks; "Intelligence without representation", Artificial Intelligence 47 (1991), 139-159.

#3 **Micro-code Memory Justification**

- A common arrangement is two levels of program memory.
  Higher or outer level is changeable and the lower or inner level is held constant.
  Raises the level of detail to a level more easily handled by the programmer.

- More than two levels are possible and often desirable. Generally it can be said that micro-code exists as a distinct level for economic reasons related to the relative cost, speed, width and size of read-only memory.

- Additional layers are placed in program memory, typically as layers or levels of subroutines. Each layer further raises the level of operation to a more abstract plateau reducing or subsuming non-relevant detail.

- Thus, the micro-code and program memory dichotomy is,
  Part of a larger picture with many levels of program

#4 **Data Stack Justification**

- Computation of state changes is expressed as algorithm.
- Algorithms are most easily expressed as factoring into simpler computations.
- This in the abstract is a nesting of functions.
- To evaluate this nest it is useful to provide a stack for partially computed operands and results.
- A stack orders this process in an efficient manner having minimal memory usage (the same memory is re-utilized for different parts of the computation).

#5 **Control Stack (Return Address Memory) Justification**

- In order to save on program memory, parts of the program are reused.
- These parts define functions and routines used (called) more than once.
- For the program to pickup where is left off a stack for the resumption point is needed.
- This state is usually the contents of the program sequence address register.
- Nesting of the control-state is conformal to that of the data.
- It is also possible to use the control stack for address calculations.
  Useful if the address size and the data size are different.
  More evenly distributes the stack operations between the two stacks.

#6 **Token Table Justification**

- The mapping form the program code to the basic machine sequences (e.g. micro-code) can be made with a lookup table.
- Allows arbitrarily designed code (arbitrary instruction sets) to be mapped into the “constant” basic machine.
- Rarely done in practice, generalizes the capability of the basic machine.
- Usually the mapping from instruction codes to micro-code is done via dedicated read-only table or by its logic array equivalent.
- In the case of computers designed for interpretation, the token table can be large (e.g. x86 emulation).

#7 **Logic Equations**

- For a FPGA chip implementation of a microprocessor:
  Equations mapped into small lookup table(s).
- Each table has a specific number of bits.
- Thus, logic can be mapped to memory.
- True information count can be determined by compressing set of all tables.
- Both theoretical and implementation bit counts.

- For an ASIC implementation:
  Enumerate set of logic gates in use
- Number of gates times log base 2 of # of gate types is bit count.
  Some compression of bit-count possible

#8 **Logic Connectivity**

- On a FPGA or CPLD chip:
  Connectivity between the RTL equations is mapped onto
  FPGA or CPLD connectivity matrix.

- Connectivity matrix is “fused” by memory:
  E^2PROM typical for CPLD
  Static RAM typical for FPGA

- If one has access to internal fuse map:
  Determine portion of chip allocated to design
  And prorate total fuse count of chip

- From the net list:
  Enumerate signals
  For each input to LUT or Gate,
  Log base 2 of signal count is theoretical minimum info bit count

- Compare CPLD and FPGA bit counts with theoretical bit counts

Miscellaneous

#9 Register space

#10 I/O space

#11 FIFOs and Buffers

#12 Nano-Manufacturing complexity

#13 Protein pattern matching

Von Neuman Architecture:

• Program	main memory
• Data	main memory
• Data stack	main memory, recursion not always possible
• Control stack	main memory, recursion not always possible
• Micro-code	main memory or hardwired
• Instruction decode	hardwired

Harvard Architecture:

• Program	program memory, possibly read-only
• Data	main memory
• Data stack	main memory
• Control stack	main memory
• Micro-code	main memory or hardwired
• Instruction decode	hardwired

“Wilkes” Architecture:

• Program	main memory
• Data	main memory
• Data stack	main memory
• Control stack	main memory
• Micro-code	read-only memory
• Instruction decode	2nd read-only memory

Stack Frame Architecture

• Program	main memory
• Data	main memory
• Data stack	main memory, part of stack frame
• Control stack	main memory, part of stack frame
• Micro-code	read-only memory
• Instruction decode	2nd read-only memory

Dual Stack Architecture

• Program	main memory
• Data	main memory
• Data stack	cached to main memory or separate memory
• Control stack	cached to main memory or separate memory
• Micro-code	program memory, sometimes separate
• Instruction decode	hardwired or tokens mapped into subroutine call

RISC Architecture

- Memory operations separate from computation operations
- Resemble micro-programmable machines
  Micro-code store is merged with the main program store
- Compiler does the micro-programming
- Current practice:
  Separate instruction and data caches

• Program	main memory
• Data	main memory
• Data stack	register window or main memory
• Control stack	merged with data stack
• Micro-code	merged with program memory
• Instruction decode	hardwired

DSP (Digital Signal Processor)

- Attempt to get the most performance per clock cycle,
  And the most algorithm per instruction
- Instruction set is often highly tuned to digital signal processing
  Both von Neumann and Harvard types exist
- On chip instruction and data memory are often provided
  Often with independent access

ASIC/PLD Architecture:

- Memories can be numerous
  And sized for use
  Width of micro-code not a problem
- RAM: ~Six transistors per bit
- Dual Port RAM: ~Eight transistors per bit
- ROM: ~One transistor per bit
  Or implemented as logic
- Gate: ~Four transistors per gate
- Function Units often pipelined
- On Chip FLASH common
- On Chip DRAM possible

References:

Computer Architecture books

l) C. Gordon Bell & Allen Newell; Computer Structures: Readings and Examples; McGraw Hill, 1971.

2) C. Gordon Bell , J. Craig Mudge, John E. McNamara; Computer Engineering: A DEC view of hardware systems design; Digital Press, 1978.

3) Gerrit Blaauw & Frederick Brooks Jr.; Computer Architecture: Concepts and Evolution; Addison Wesley, 1997. (has extensive bibliography)

4) Philip Koopman; Stack Computers; the new wave; Halstead Press, 1989.

5) Simon H. Lavington; Early British Computers; Digital Press,1980.

6) David A. Patterson & John L. Hennessy; Computer Architecture: A Quantitative Approach; Morgan Kaufmann, 1990.

7) Alan B. Salisbury; Microprogrammable Computer Architectures; American Elsevier Publishing, 1976.

8) Daniel Siewiorek, C. Gordon Bell & Allen Newell; Computer Structures: Principles and Examples; McGraw-Hill, 1982.

9) Stephen Ward & Robert H. Halstead; Computation Structures; MIT Press, 1990.

10) Bruce Shriver & Bennett Smith; The Anatomy of a High-Performance Microprocessor, A Systems Perspective; IEEE Computer Society Press, 1998. **(Attached CDROM contains many of the classic papers)**

Journal articles

11) Tom Hand; The Harris RTX 2000 Microcontroller; The Journal of FORTH application and research; 6:1:5-13, 1990. (16-bit, dual on chip stacks)

12) John Hayes & Susan Lee; The Architecture of the SC32 Forth Engine; The Journal of FORTH application and research; 5:4:493-506, 1989. (32-bit, dual cached stacks)

13) Makoto Hasegawa & Yoshiharu Shigei; High-Speed Top-of-Stack Scheme for VLSI Processor: a Management Algorithm and Its Analysis; SigArch 13:3:48-54, 1985(12th Annual Symposium on Computer Architecture). (study of stack caching schemes)

14) M. V. Wilkes; The Genesis of Microprogramming; Annals of the History of Computing, 8:116. (reprint of "The best way to design an automatic calculating machine", also reprinted in Siewiorek, Bell, & Newell; Computer Structures: Principles and Examples, McGraw Hill, 1982)

15) R. E. Prather; Comparison and Extension of Theories of Zipf and Halstead; The Computer Journal, 31:3:248-252, 1988. (token statistics)

Web Sites

16) www.ptsc.com has a 32-bit dual stack chip, stacks overflow to main memory

Philosophy:

There are several levels at which one can examine a computer's architecture. There is an implementation level spelling out particular register and ALU resources. And there is a programmer's machine language level at which resources are made more "logical" than "physical". This paper tends toward the programmer's level.

The RISC architectures with large register sets rely on a compiler to convert the physical level resources into logical level allocations. This can be considered an advantage for the RISC architectures when using compiled languages. The compiler optimizes resource allocation within the context of the language and a given program.

My interests lie in a more direct mapping between hardware resources and the programmer's resources. There are two application areas where this is necessary: real-time where knowledge of execution times is important; and interpreters where the software factoring or decomposition structure is adjusted to achieve a good matching between the algorithm and the hardware.

A Historical Tour:

The first stored program computers were of two types: the Harvard architecture had separate memories for program and for data. Generally the data memory also contained any stacks and the micro-code and op-code tables were hard wired. The von Neumann architecture combined data and program memory. Although the simplicity and potential power of the unified data and program memory are considered advantageous, separate memories are the norm for DSP chips and often the case for caches.

The next development was the "Wilkes" architecture. It was possible to implement micro-code in a fast read-only form. One perspective on micro-code is that it is a set of low level subroutines raising the hardware resources from a fairly primitive level up towards a programmer's level. Depending on various design decisions the micro-code can be very low level or can be at the RISC level. In any case micro-code is often used where an operation takes several clock or memory cycles.

The stack machines where an attempt to make the hardware fit the mold of recursive languages. When the stacking levels are unpredictable and potentially large then support for memory based stacking mechanisms are usually included.

The Algol programming language and its descendents typically implied a memory model using merged data and control stacks. It was considered ideologically favorable to encode each level of recursion as a stack frame complete with parameters, locals, and return address.

The idea of a stack frame interferes with the stack operators of a true stack based computer. A true stack machine uses top-of-stack operators and rearrangements. A return addresses on the data stack in the midst of the parameters and results would get in the way. Thus, on true stack machines a separate control or return address stack is used.

Many of the micro-code controller chips have a small stack for micro-code return addresses. Some of the true stack based chips have small data and return address stacks. Stack caches are another way to implement stacks and show good economy (a small, typically 16 register, stack cache automatically managed by the microprocessor reduces memory traffic dramatically).

The RISC advocates would argue that it is better to use a simple set of fast registers and a register allocating compiler than to implement a data or control stack or stack caches. They claim that theirs is the fastest, most efficient approach. There is little argument that a fast stack architecture is more difficult to design and requires greater external signal count or greater internal data paths (although the existing stack architecture chips have low gate counts and fairly fast clock rates). The RISC approach assumes an optimizing compiler. The cost of the compilation step becomes a factor in interactive environments and thus argues for better match between the instruction set and the programming language.

The last memory is that for op-code vectoring. On a small simple machine this is often hardwired. On a micro-coded architecture this is often a small fast lookup table. Some machines have been designed to support several instruction sets and here this table is a small fast RAM. The op-code table is considered a memory space in its own right on the possibility that a large programmable instruction set is desired. This is most likely to be the case with a computer designed to serve as an interpreter. Here the tokens to be interpreted are vectored into micro-code or main memory addresses. For this reason it is called the token table. The ability to vector to either main memory or micro-code memory would allow the token usage statistics and micro-code table size to be traded-off.

Examples:

The classical von Neumann architecture culminated with the Accumulator design of the 1970s. There was an accumulator, an accumulator extension, and several index registers. Index registers were designed to speed array addressing. Return addresses were placed in memory with the routine called. These were Fortran engines.

If the instruction set was complex, most likely it was implemented via micro-code. Micro-code allowed large complex instruction sets with the aim of lowering instruction bandwidth and optimizing hardware throughput.

The Harvard architecture continues, most strongly in DSP (Digital Signal Processor) chips. Instruction word length and data word length can be and often are different. The two separate memory paths operate independently thereby doubling memory transfer rate.

The early stack machines were designed to support recursive programming. The stack was allocated in frames where each frame contained the parameters, the working or local variables, and the return address.

The register based machines unified all of the above by making all the registers able to hold either data or addresses, allowing access to any register, and by adding addressing modes for software stacks. Basically this is the current era.

The RISC machines attempt to speed the register-based architecture by using more registers, pipelining instruction processing, and separating memory load/store from arithmetic. With current technology this all fits on one chip. The program memory bandwidth requirement is higher and is assumed not to be a limiting factor (typically there is both an instruction and a data cache). RISC machines can be considered compiler micro-programmed architectures.

A few dual stack machines exist. Since the emphasis is not on the stack frame but on actual stack operators these are true stack machines (instead of register load and store, there or operators for duplicate, drop, swap, etc.). Subroutine calls are very common in code for these machines. (example: Patriot Scientific's "Ignite" 32-bit chip, www.ptsc.com)

Often subroutine return is signaled by a dedicated bit in each instruction and as such the return mechanism occurs in parallel with the rest of the instruction. Thus the dual stack machines have lower subroutine overhead than the frame based designs (subroutine return is usually "free" and no extra frame setup is done).

The shortness of subroutine code (as little as one instruction) allows "intelligent" data structures (where code is embedded with data). As code generation is straight forward, the large compiler/expert system is not required. In Forth, the compiler is actually distributed and user modifiable.

A remaining architecture class with very few examples and no current implementations is the “token” machine. Here the instruction encoding is subject to programmer manipulation. Thus there is a lookup table to convert tokens to micro-code or instruction starting addresses. The Burroughs 1700 and the Nanodata machine are the two main examples. Presumably the difficulty of supporting many instruction sets was greater than the efficiency obtained from increased resource utilization (which was the main selling point).

Comments:

The RISC architecture with load/store instructions separate from arithmetic instructions is not new. It occurred in several early micro-programmable machines, the Data General NOVA series, and the CDC and Cray supercomputer series.

Each separate memory implies increased memory bandwidth. Thus the von Neumann architecture has the lowest performance. A separate wide-word micro-code memory dramatically adds instruction bandwidth. The stack machine offers up to five independently operating memories. The token machine offers a sixth independent memory path.

In current technology splitting data and program offers the biggest first gain in memory bandwidth. Micro-code memory at one time offered considerable gain but now with pipelined (no micro-code) floating point the gain from micro-code is considerably less. The data and control stacks offer additional bandwidth but with a large register set and a smart compiler these can be emulated. Where the compiler can not optimize the code (as in interpreted code) there is a greater gain from such stacks. Finally where the interpretation is under programmer control it is worth while to speed the token lookup with a dedicated jump table.

The number of stacks needed for a stack architecture computer has no upper limit. Some have proposed separate stacks for each type of data, such as floating, integer, string, and address. On the control side, besides return addresses there are: loop, if, and case statement back pointers.

Most modern computers (anything dating after the PDP11 series of Digital Equipment

Corp.) have addressing modes supporting stacks via indirection with post auto-increment and pre auto-decrement. Thus one can allocate as many registers as needed (more than half of the available registers could be so used) for different stacks.

Subroutine call/return overhead continues to be a problem even on the fastest RISC chip. This results in a tendency for small subroutines to be avoided (i.e. expanded inline). The dual stack machines are an exception and generally employ smaller subroutines. Thus dual stack machines allow more general code factoring. This also results in smaller programs.

The "token" architecture where the computer processes or interprets a stream of commands is a natural mode for several types of data: compacted or compressed data, tokenized code, and user defined instruction sets. The tokens generally follow a "1/n" population distribution (the second most common token occurs half as often as the most common token). Given this distribution, a fixed token size (log base 2 of number of tokens) is fairly space efficient and easily hardware efficient. Each such token set could be of a different bit length implying that token processing operates at arbitrary bit boundaries. A variable size encoding would, of course, require a variable displacement shifter in the instruction decode section.