Addison Architecture

by Brian McMillin

Contents

Abstract

The Addison Computing Architecture is a theoretical architecture for secure, general-purpose applications.

Key features include:

- Bit-oriented memory addressing
- Frame-based rather than Stack-based operations
- Descriptor-based memory and register addressing
- Program and Memory addresses isolated from user-data
- Virtual registers implement traditional ALU operations
- Program instructions consist ONLY of data move source to destination
- Branches and Conditionals are separate from program instructions
- Virtual and Physical memory allocation are distinct and inherently safe
- Hardware support for data structures via “subscripting” operations

It is intended that actual hardware implementations may incorporate a subset of these features. Experience gained through subset choices will help determine which features generate the best performance in the real world.

Unfortunately, hidden architectural assumptions buried deep within many current algorithms make it quite difficult to make off-the-cuff performance comparisons. Much effort will be required to generate meaningful benchmarks.

Rationale

Hardware architectures should support the computational operations required to solve specific problems. In many cases the description and analysis of the specific problem is prematurely influenced by features of current philosophies of hardware or software design. Data structures and algorithms may be chosen more because of experience with traditional methods than by the actual needs of the application.

The Addison Architecture is an attempt to separate the algorithms and data structures from hardware considerations such as memory allocation, word sizes and control flow.

In addition, this architecture inherently prevents common programming errors and security breaches with its secure design philosophy. It is conceptually impossible to access unallocated memory, execute data as code, manipulate memory pointers or overflow a stack.

The philosophy is

- Do what I **WANT** to do.
- Do not force me to do things I don't **NEED** to do.
- Do not **ALLOW** me to do what I should not do.

The architecture provides specialized hardware support for features that are

- **Valuable** - bit fields, subscripting, parameter frames, bounds checking, etc.
- **Possible** - specialized, dedicated hardware can be designed and built
- **Fast** - independent, parallel operation in separate hardware modules

Although this architecture seems complex, the net effect should be operation that is both **FASTER** and **SAFER** than more traditional approaches.

Background

Over the past 75 years the concepts of digital computing have explored many avenues. The early work of Babbage and Lovelace has developed a firm mathematical underpinning and electronic integrated circuits have enabled massive improvements in speed and complexity.

After much experimentation in a variety of areas a general consensus has emerged for many aspects of computer design. Successive refinements in each area have resulted in very high performance.

The selection and trade-offs between each of these areas is the province of computer architects looking at each specific application today.

I present a brief analysis of several of these areas. I contend that the successive improvements in each of these areas has resulted in peak system performance that is only a LOCAL MAXIMUM. In order to achieve radical improvements in performance, significant departure from this hard-earned state-of-the-art will be required.

- Address Spaces
- Registers
- Instruction Sets
- Control Flow
- Memory Allocation and Virtualization
- Multi-Tasking
- Interrupts, Faults, Traps and Exceptions
- Parameter Passing
- Objects, Structures, Methods, Generators
- Data Structures: Constants, Scalars, Vectors, Lists and Trees
- Protection: Read, Write, Execute, Protected, Allocated, Bounded
- Software

Software

Software development and programming style has traditionally been strongly influenced by hardware architecture. Programmer's knowledge of the features of a particular hardware implementation has colored all aspects of their designs. This is natural, since they would want to make use of wondrous features in an attempt to achieve maximum performance. Unfortunately, this premature optimization means that most modern algorithms address not the Actual Problem but the constraints of ancient hardware conceptually embedded in thought processes and development tools.

Address Spaces

Turing Machines are theoretically capable of performing any computing operation, albeit without regard to time or memory usage. Optimizations and concessions to the real world have resulted in a variety of different approaches to making computation economically feasible.

Von Neumann machines anticipate that program instructions and data may be manipulated by slightly different hardware working on the same shared memory space.

Harvard Architecture machines separate the program instructions from the data storage as a technique for simplifying and parallelizing memory accesses.

Stack-oriented architectures try to simplify accesses to frequently-used data such as function parameters and local variables. When fully implemented, true stack architectures can eliminate addressable CPU registers altogether.

Descriptor-based architectures use hardware protection mechanisms to distinguish address pointers from user data. Additionally, array bounds-checking and execution-protection features are usually available.

Registers

All computer architects must confront the existence of a wide range of memory speeds, capacities and costs. Many approaches to this problem have been tried but they generally boil down to creating a many-layered hiererchy of speeds and sizes.

The fastest memory is allocated as a set of multi-bit registers that are immediately accessible by the processor computing elements. The choice of size and number of these registers is fundamental to an architecture.

The choice of register size directly affects the scale of computations performed by the hardware and the bus-width of data transfers throughout the memory hierarchy. The number of registers generally affects the rate at which access to memory is needed.

Many different register size and number combinations have been tried over the years. The general trend has been to make register lengths an ever-increasing power of two.

Machine	Register Size	Number of Registers

Intel 4004	4	16
Intel 8008	8	8
DEC PDP-8	12	4
DEC PDP-11	16	8
ARM Cortex	32	15
DEC PDP-10	36	16
Burroughs 5000	48	2
CDC 6000	60	8
Sparc	64	31

Instruction Sets

The general concept of Complex Instruction Set machines is that the hardware will perform a (sometimes arbitrarily) large set of internal operations over many clock cycles in the course of accomplishing the goals of a particular processor opcode.

In contrast, the introduction of Reduced Instruction Set architectures envisioned the use of a rather small set of very fast, generally single-cycle instructions that could be combined sequentially to accomplish the required operations.

Micro-coded processors implemented complex instructions as a series of simpler operations controlled by a look-up table. The CISC opcode would index this micro-code table, which would then provide the control lines and timing required to perform the operation.

Micro-coding simplified the instruction decode logic, and allowed an approach to what was later deemed “very long instruction word” processors.

The Data General Eclipse (and others) implemented a Writable Control Store. The WCS allowed dynamic changes to the instruction set by rewriting the micro-code table for selected instructions as a part of normal operation. The WCS allowed specialized applications to be executed on semi-custom hardware - sometimes with great speed advantages. However, the difficulty of managing the WCS in a multi-programming environment severely limited its utility.

Bounds Checking

Early processors were constrained by the number of gates that could be built onto a die. This meant that the Arithmetic Logic Unit (ALU) represented a large part of the gate budget for the entire chip. Designers realized that the single ALU could be used for all sorts of clever things. Not just arithmetic and flow control for the user's program, but address generation for pointers to data in memory.

Instead of having dedicated address-related hardware, the (single) ALU and register set was used to sequentially
* compute memory addresses,
* verify array bounds,
* and access data in memory.

The lack of dedicated hardware for memory address generation and validation slowed the effective speed of the program, and led compiler designers to create ill-advised options to "Turn Off Bounds Checking". This definitely speeds up the program. It makes managers and customers happy. And we all know that programs never break after we have managed to run them successfully in the lab a few times…

As the demands increased for systems with virtual memory and multi-programming it became obvious that hardware memory protection would be necessary. This was implemented as simply as possible by making fixed-size pages and assigning each page certain priviledges such as read-only or executable.

This protected against many gross problems, but did nothing for the subtle errors that lead to crashes and malicious exploits.

The mindset that treats memory addresses as "just another number", allows user programs indiscriminate access to them in the form of pointers and allows them to be randomly scattered throughout various data structures, means that truly safe and protected programs are not possible.

Multiple-function Functions

Frequently programmers are tempted to create software functions that are capable of performing several different operations depending on the value of one of the parameters.

```
function incOrDec(fInc, value) {
    if (fInc) {
        return value+1;
    } else {
        return value-1;
    }
}
```

Unlike the contrived example above, this type of programming is all too common in the functional APIs of operating sysems and function libraries.

The normal explanation for this is usually that there are logically related common features (perhaps validity checks) shared among the different options. Also, making many different, tiny functions, each with a different name, is perceived as confusing or otherwise poor practice. Or the CS-101 instructor said you had to demonstrate a use for **ENUM** constants.

In any case, it is my contention that there is no valid use case for passing a constant to a function. For clarity and legacy applications, the program source may *appear* to pass constants, but in all cases the compiler should factor out the constants and *actually* make calls to faster, pre-parsed functions that make no use of such parameters.

Another classic case is the Operating System Call function. Often this is implemented as the handler for a (single) Software Interrupt instruction.

```
function sysCall(kind, args...) {
    switch (kind) {
        case kOpen:  doFileOpen (args); break;
        case kRead:  doFileRead (args); break;
        case kWrite: doFileWrite(args); break;
        case kClose: doFileClose(args); break;
        ....
    }
}
```

The point here is that the programmer and the compiler know at a very early point exactly which function ultimately needs to be invoked, but inserts an entirely artificial layer of complexity into the actual code. In this case, the claim is usually that the System Call instruction (interrupt) is necessary to change the priviledge level for the system, making the Operating System code more “trusted” than the user code.

In the Addison Architecture, priviledge is a property of individual descriptors. The hardware rigidly controls the creation, duplication and destruction of descriptors. All memory descriptors are created as subsets of a previously-held descriptor. Thus, access to arbitrary memory, execution of arbitrary code and artificial escalation of priviledges are prevented by design.

Even apparently priviledged functions (such as the program loader itself) which are capable of manipulating descriptors and allocating memory are constrained by the hardware to areas already allocated to them. This means that allocations are hierarchial and completely independent of one another.

Independent, hardware-enforced allocations mean that all processes and structures are inherently protected from corruption or improper actions by any other actor. The actual concept of an Operating System becomes essentially synonymous with any other process.

Overview

The Addison Architecture can be considered an example of a new class of computer that has no instruction-set at all in the traditional sense. Instructions consist solely of pairs of references Source and Destination data descriptors. Essentially, the only “instruction” is MOV S,D.

This is made possible by implementing three features:

- A Function Frame contains the set of Descriptors that can be referenced
- Branch/Conditional information is separate from the actual instructions
- Arithmetic and Logical operations are invoked through the use of Virtual Registers

Implementation

An actual implementation of the Addison Architecture is described. This documentation should provide enough depth to build a simulator of a working system. This simulator will allow examination of potential performance. It will also provide a basis for designing and testing the required compilers and other software development tools. A high-fidelity simulator will also enable design of physical processor hardware.

Parameter Frames

Parameter Frames contain all the information necessary to describe the data and code required for a function. This information is contained in a set of descriptors that are subscripted by the hardware during function call and return, and by the Source, Destination pairs during code execution.

A hardware register (FP) contains the base address of the current Parameter Frame.

Data Descriptors

Data Descriptors are structures that encapsulate the necessary information to make a safe pointer to bit-fields in memory. In addition, a Data Descriptor may refence a virtual register. This means that subscripting into the list of Data Descriptors provides the shorthand notation of Source and Destination used in the Instruction Code.

It also means that every memory reference is constrained and verified. Out of the universe of all possible references, only certain meaningful (as understood by the compiler) references are allowed at any given time.

As a conceptual minimum, consider that a Data Descriptor includes:

Logical Address	Logical Length	Record Length	Record Subscript	Field Offset	Field Length
..					

The actual memory address is computed as:

```
Logical Address + (Record Length * Record Subscript) + Field Offset
```

This structure allows hardware Exceptions to be generated for references to:

- Unallocated memory
- Deallocated memory
- Fields outside the Allocated memory area

To be valid, a reference must meet the conditions that:

```
Logical Address > 0
Logical Length > (Record Length * Record Index) + Field Offset
```

Remember that in the Addison Architecture all addresses, fields and lengths refer to single-bit units.

It is perfectly fair, meaningful and safe to do a memory allocation of a single bit. Although (perhaps) a bit extravagant with the overhead.

Setting the Record Subscript field in the descriptor is a User-Mode operation that sets up the Descriptor for an actual memory field access. Hardware implements the required multiply-and-add, and sets internal flags associated with each descriptor. Thus, the possible exceptions can occur immediately upon reference, and do not occur during the Subscripting operation.

Physical Allocation Descriptors

In a Virtual Memory implementation, each Data Descriptor also has an associated list of zero or more Physical Allocation Descriptors. These contain

As a conceptual minimum, consider that each Physical Allocation Descriptor includes:

Physical Address	Physical Length	Logical Address	Logical Offset
..			

The Physical Allocation Descriptor allows mapping of the previously described Logical Address space onto physical memory. It includes the addition of Memory Fault conditions which, unlike Exceptions, are handled by the operating system. This makes allocation and swapping of physical memory a distinct operation from the allocation of logical memory.

Encountering a memory fault condition is a resumable operation: if the operating system is able to make the required memory available it updates the Physical Allocation Descriptor and the program continues with the fault condition corrected.

Constants

Constant bit fields and structures are generated by the compiler and accessed through descriptors marked as read-only. The actual memory is part of the Code Frame so that Constant data can be shared across many Parameter Frames

Scalars

Scalars are indicated by Data Descriptors with a zero Record Length.

Vectors

Vectors are the representation of traditional arrays. Setting a Record Subscript value for the Data Descriptor selects one of many field elements. Unlike simple arrays, the size of the record used for Subscripting is independent of the size of the data field being referenced.

In C-language terms, this is an implementation of an array of struct.

Subscripting allows a program to directly set the Record Subscript of a Data Descriptor. Subscripting is an application-level operation involved in performing a particular algorithm.

Indexing is an operation involving physical or virtual addresses or offsets and is not accessible by the user program.

In addition to directly setting the Record Subscript as part of program operation, a Data Descriptor may be configured to automatically increment or decrement the Record Subscript. This allows the program to sequence through vector elements without the need to perform arithmetic on the Subscript in many cases.

Lists

Lists are doubly-linked lists of records with links maintained by the hardware. Subscripting causes sequential traversal from the head to the specified element. Incrementing or decrementing the Subscript is accomplished by following a single link.

Insertion and deletion of records are supported.

Sequential searching of a list can be performed in a single instruction that repeats until the condition is met.

Trees

Trees are triply-linked structures of records that support direct subscripting and incremental traversal.

Insertion and deletion of records are supported.

Binary searching of a tree can be performed in a single instruction that repeats until the condition is met.

Parameter Frame Optimization

The Addison Architecture relies on the Compiler/Linker/Loader toolchain to analyze the flow control and function-calling patterns of the generated program. In particular, flow control operations are handled separately from data manipulation instructions.

Importantly, Parameter Frames are used to store all input parameters, output results and local variables for all functions. The required frames are established statically during program or thread initialization.

Dynamically created frames may be used to support recursion, but this is strongly discouraged.

Static Parameter Frame allocation eliminates the need for a program stack. This is advantageous because the continual creation and destruction of stack frames is wasteful of time and does not contribute to the performance of the algorithm. Pointers to parameters on the stack cannot be sanity-checked and lead to many security problems. In addition, the arbitrary-sized allocation of space to the stack is wasteful of memory - especially when many threads are required.

Consider the example:

```
function f() {
    var a, b, c...
    for (var i=0; i<9; i++) {
        g(i, a, b);
        h(c, i);
    }
}
```

The stack frames for the calls to functions g() and h() are not only allocated outside the loop, they are allocated separately by the compiler and initialized by the linker.

The pre-allocated frames match the calling *tree* of the functions.

Now, consider the example:

```
function f() {
    var a, b, c...
    for (var i=0; i<9; i++) {
        g(i, a, b);
        g(i, b, c);
        g(a, b, c);
    }
}
```

There may be unique, pre-allocated stack frames for each of the distinct calls to function g().

The pre-allocated frames match the calling *pattern* of the functions.

Code Frames

Program code consists of instructions that move and manipulate data in memory and flow control that establishes the sequence of these operations. Traditionally, flow control operations (tests, branches, calls, returns, etc.) and parameter passing have simply been treated as instructions and executed sequentially along with all other code.

This is a philosophical approach that has led to many elaborate hardware designs that attempt to improve performance using stack push and pop, variable-length or prefixed instructions, branch prediction logic, etc.

The Addison Architecture recognizes these as distinct entities: Parameter Frames, Flow Control and Operation Sequences.

Each Code Frame contains the following elements

- Branch / Conditional Table
- Instruction Code
- Constant Data Structures

Parameter Frames are initialized by the Compiler/Linker/Loader as part of the executable program. Each parameter frame refers to a Code Frame. The Code Frame contains the Branch/Conditional Table followed by the Instruction Code sequence of Source, Destination

pairs.

It is important to recognize that there may be many Parameter Frames that refer to a given Code Frame. This allows code reuse for multiple-thread reentrancy, recursion, and the unique Parameter Frame Optimization feature of the Addison Architecture.

Constant Data Structures, if needed, are part of the Code Frame and are shared by all invocations of the associated functions.

Branch / Conditional Table

Think of the Branch / Conditional Table as an example of traditional object code with all of the non-branching operations removed. This becomes a compact representation of the flow-control pattern of the algorithm.

Combine this with the conditional feature of the Addison Architecture that leverages the Virtual Register concept to make all comparisons and inherent property of the register set. Specifically, each register pair has a set of comparison bits that indicate the relationship of their current contents. Thus, the conditionals are automatically evaluated each time data is stored in a register. These conditionals are referenced by the branch descriptors.

The Branch / Conditional Table is a list of Branch Descriptors.

Trigger IP	Condition	Target IP or Parameter Frame	New BCT Subscript

During operation, instructions are fetched from the Instruction Code section as pointed to by the IP. The BP points to an entry in the BCT that indicates the next conditional in the code sequence. When the sequential execution of instructions causes the IP to reach the TriggerIP, the possibility of a branch exists.

If the Condition in the current BCT entry is fulfilled, a branch will occur. Branches in this sense may be either simple transfers of control within the current Code Frame and Parameter Frame, or they may be Calls which change either or both frames.

If the Condition is not met, the BP is incremented and the next BCT entry is loaded. Execution continues with the incremented IP.

Possible Conditions:

Condition	Description
Always	Unconditional Branch
a < b	Less Than
a <= b	Less Than or Equal To
a == b	Equal To
a >= b	Greater Than or Equal To
a > b	Greater Than
a != b	Not Equal To
Never	Continue sequential execution, but change Parameter Frame

Instruction Code

Instruction code consists of a sequence of Source/Destination pairs. Essentially, the only opcode in the Addison Architecture is MOV S,D.

The Source and Destination in the instructions are short binary integers that are subscripts into the descriptor table of the current Parameter Frame.

Source	Destination

The concept is that the Parameter Frame contains descriptors to any of the memory areas or virtual registers that are relevant at this point in the program. Since the relevant set is always tiny compared to the universe of possible addresses we have an implicit program compression feature. Smaller instructions mean that a larger number of program instructions can be loaded with each memory reference.

Further, the number of bits in the Source and Destination fields may be adjusted based on the needs of the algorithm or implementation. The size of the parameter frame generally dictates the size required by the Source and Destination fields. This may be fixed by the hardware design, or a feature adjusted on a frame-by-frame basis.

Parameter Frame Switching

The important goal of the frame-based architecture is to allow for accurate pre-fetch of required data well in advance of its actual use.

The knowledge of the actual program flow known by the compiler toolchain is leveraged to directly improve program performance.

Calls

A Call is performed when the Target field in the current Branch / Conditional Table entry points to a new Parameter Frame. The Entry Code address new Parameter Frame contains the address of the new Code Frame.

The call operation proceeds as follows:

1. Determine that the current instruction pointer matches the IP field in the current Branch / Conditional Table entry.
2. Determine that the Condition specified in the current Branch / Conditional Table entry has been met.
3. Determine that the Target specified in the current Branch / Conditional Table entry is the address of a Parameter Frame.
4. Compose a Return Address value and store it in the new Parameter Frame.

The Return Address Value consists of the addresses of the current Parameter Frame, the current IP (updated to the next instruction), the current Branch Pointer (also updated for the next instruction).

1. Load the new Parameter Frame base address.

2. Load the new Code Frame base address.
3. Load the new Instruction Pointer.
4. Load the new Branch Pointer.

The processor then fetches the next instruction and Branch / Conditional Table entry. The call to the function is now complete.

Returns

A return is indicated by a null Target in the current Branch / Conditional Table entry. The Instruction Pointer, Branch Pointer, Parameter Frame and Code Frame are loaded from the Return Address Value in the current Parameter Table. This effectively returns the processor to the instruction following the Call.

Exceptions

An Exception can occur on any instruction. One example of an exception is a memory access via a Descriptor that points outside the valid Logical Address range.

Exceptions cause a branch to the location indicated by the Exception Address Value in the current Parameter Table. Exceptions are not Calls, and return information is not saved.

The Instruction Pointer, Branch Pointer, Parameter Frame and Code Frame are loaded from the Return Address Value in the current Parameter Table.

Traps

A Trap occurs when Unimplemented Instructions are encountered. The Virtual Register concept allow for operations to be implied by access to register addresses that may not correspond to physically implemented registers. Simple examples such as adding or incrementing would likely be implemented in hardware. The concept can be extended to more complex operations. Mathematical operators such as SQRT or SIN might be implemented in specialized hardware on certain processors. In order to maintain generality of code, attempting such instructions on a processor without the specialized hardware will result in a Trap.

Traps can be viewed as Calls which do not have an entry in the Branch / Conditional Table.

Operating System Calls can be implemented as Traps by using references to unimplemented Virtual Registers in the Parameter Frame.

Faults

A Fault occurs when an instruction will attempt to access memory via a Descriptor that points outside the valid Physical Address range. Faults are part of the Virtual Memory architecture and allow a memory manager to transparently map a Physical Memory block to the Logical Address space being used by a program.

Faults can be viewed as Calls with a Return Address to the current IP - not the IP of the next instruction. This allows the current instruction to be re-tried after the Fault Contition has been corrected.

The Fault mechanism is used to implement the Memory-Mapped File I/O system.

Interrupts

Interrupts can occur between any instructions. Interrupts are asynchronous occurrences, independent of the instructions or data being handeld at the time.

Interrupts can be viewed as Calls with a Return Address to the current IP - not the IP of the next instruction. This allows the current instruction to be executed after the Interrupt is processed.

Real-Time Clock

Periodic interrupts allow the Operating System to perform preemptive task switching. The interrupt interval is expected to be programmable and a function either of wall-clock time or processor cycles.

I/O Subsystem

Interrupts are expected to occur when an I/O operation is completed. The Operating System processes Queued I/O requests as they complete.

Inter-Process Communication

Message handling is an Operating System function handled in a manner similar to I/O Subsystem operation.

Virtual Registers

Twos-Complement vs. Signed-Magnitude. Pros and Cons.

Load vs. Store

Complex or Application-Specific Function Implementation

Operating System Features

Bootstrapping

Memory Allocation

Processes and Threads

Input and Output

Time, Process Scheduling and Performance Measurement

Debugging Support

Glossary

Code Frame Pointer (CFP) Base Address of the current Code Frame

Parameter Frame Pointer (PFP) Base Address of the current Parameter Frame

Parameter Frame - Memory block containing Entry Descriptor, Return Descriptor and a list of Data Descriptors.

Code Frame - memory block contining a Branch / Conditional Table and Instruction Code list.

Branch / Conditional Table - List of Branch Descriptors pointed to by the Code Frame Pointer (CFP). Indexed by the Branch Pointer (BP)

Branch Pointer

Branch Descriptor - Descriptor indicating the conditions under which execution will be non-sequential.