Architecture of LISP Machines

Kshitij Sudan

March 6, 2008

A Short History Lesson …

Alonzo Church and Stephen Kleene (1930) – λ Calculus

(to cleanly define "computable functions")

↓

John McCarthy (late 60's)

(used λ Calculus to describe the operation of a computing machine to prove theorems about computation)

↓

MIT → *"Knights of the Lambda Calculus"*

↓

MIT AI Lab (~1970's)

↓

Symbolics and LMI

“MacLisp” family Machines

1975	The **CONS** prototype (MIT)	
1977	The **CADR** aka MIT Lisp Machine (MIT)	
1980	**LM-2** Symbolics Lisp Machine, repackage **CADR**	**LMI Lisp Machine** same as CADR
1982	**L-Machine -** Symbolics 3600, later 3640, 3670	
1983	**LMI Lambda**	**TI Explorer** same as LMI Lambda
1984	**G-Machine -** Symbolics 3650	
1986	**LMI K-Machine**	
1987	**I-Machine,** Symbolics XL-400, Macivory I	**TI Explorer-II -** u-Explorer
1988	Macivory II	
1989	**I-Machine,** Symbolics XL-1200 , Macivory III	
1990	XL1200, UX-1200	
1991	MacIvory III	
1992	**Virtual Lisp Machine** (aka Open Genera) I-machine compatible, running on DEC **Alpha**	

Agenda

- History of LISP machines.
- Semantic Models.
- von Neumann model of computation.
- Programming language to m/c architecture.
- Architectural challenges.
- The SECD abstract machine.
- A brief case study.

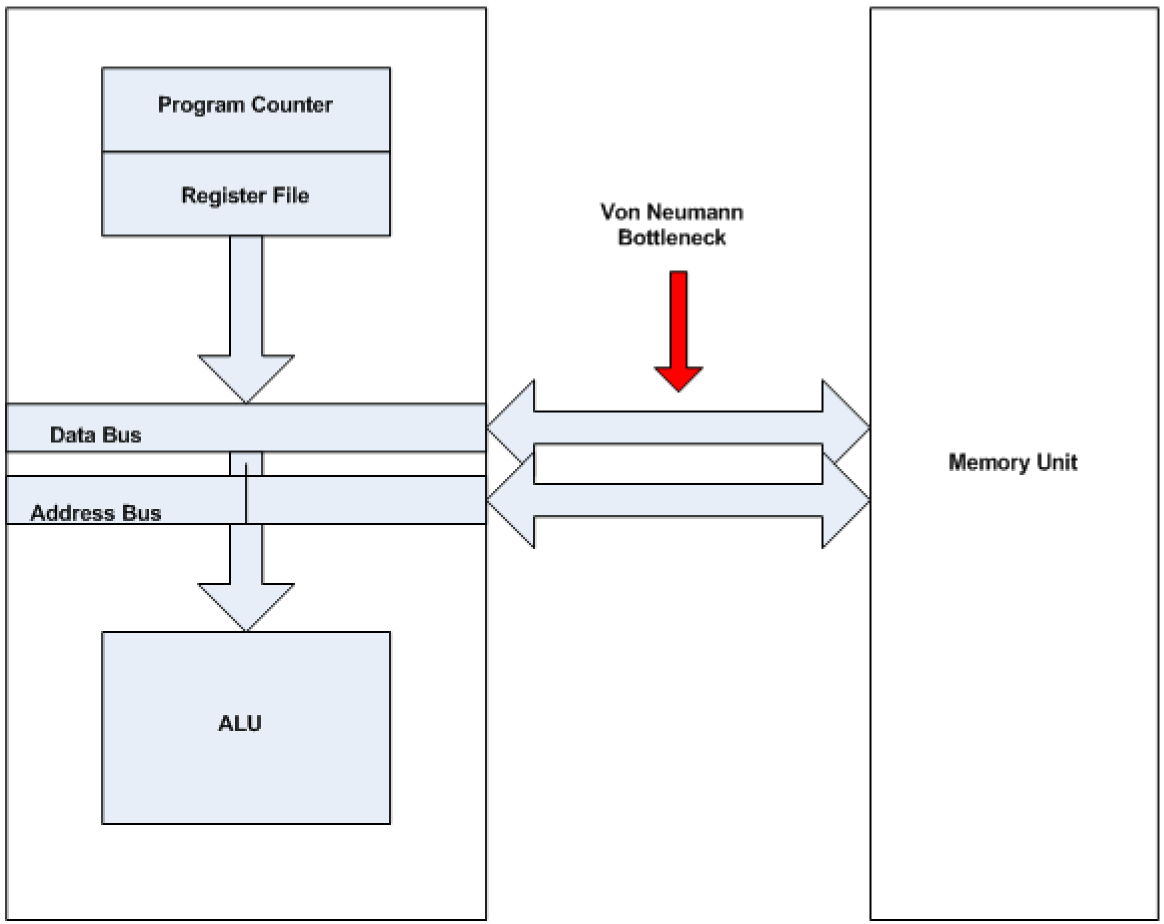
Semantic Models

- The ***semantics*** of a piece of notation is it's ultimate meaning.

Imp. for programmer→ Imp. for language designer → Imp. for architects

- Three major methods to describe and define semantics of programming languages:
  - ***Interpretive*** : meaning is expressed in terms of some simple abstract m/c.
  - ***Axiomatic*** : where rules describe data values given various objects before and after execution of various language features.
  - ***Denotational*** : syntactic pieces of program are mapped via evaluation functions into the abstract values they denote to humans.

von Neumann Model of Computation



Programming Languages to Machine Architectures

- Interplay between h/w (m/c org.) and s/w (compilers, interpreters, and run-time routines) needs to be sustainable for efficient computational structures.
- Mathematical framework → Computing models → languages → architecture → real implementations.
- Mathematical framework → ***Abstract m/c*** → real implementations.

A short detour …

- Processing symbols *"was"* touted (circa early 90's) as future of computations (*obviously hasn't happened yet!)*
- For processing symbols, declarative languages were put forth as the solution –
  - ***function-based*** and ***logic-based*** languages

So what is the future?

Architectural challenges - I

- Today we talk mostly about LISP machines (functional language m/c's).
- Describe features "*needed*" for ***efficient*** LISP program execution (RISC can obviously execute LISP).
- Language feature driven architectural hooks – we talk about then briefly.
- Abstract m/c → case studies

Architectural challenges – II

(Architectural support for LISP - I)

- Fast function calls.
  - ***call*** and ***return*** instructions with short execution latencies for dynamically bound contexts (latest active value bound to a variable name).
  - ***funarg problem.***
- Environment maintenance.
  - ***shallow- bound*** *(linked-list)*
  - ***deep-bound*** *("oblist" == global symbol table)*
    - with possible ***caching of name-value bindings (value cache).***

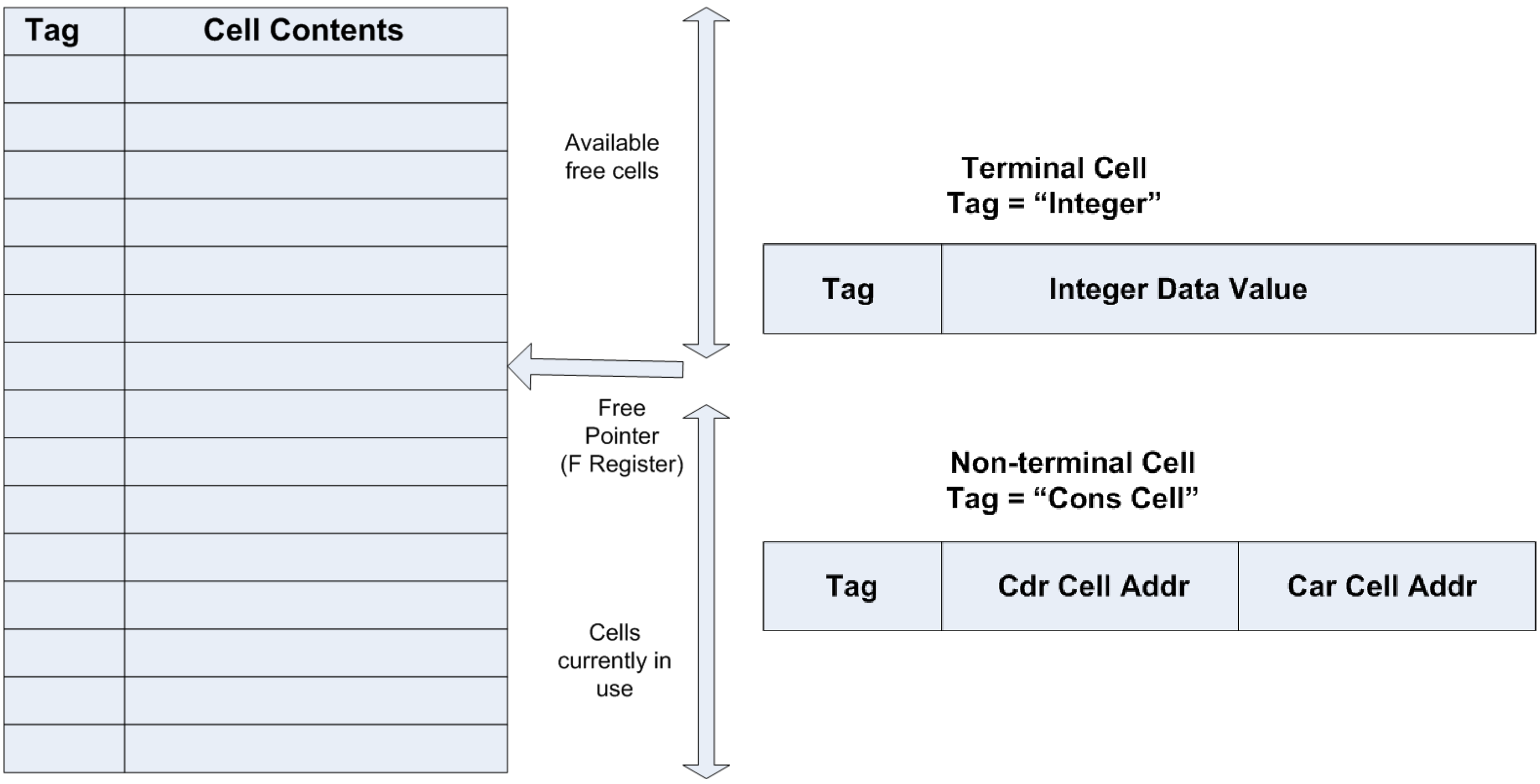
Architectural challenges – III

(Architectural support for LISP - II)

- Efficient list representation.
  - improvements over ***two-pointer list cells***
    - ***Vector-coded*** *(represent linear lists as vector of symbols)*
    - ***Structure-coded .***
      - *each cell has a tag for it's location in the list.*
      - *associative search leads to fast access.*
- Heap maintenance (a.k.a. garbage collection)
    - ***Marking*** *(accessible lists "marked", others reclaimed)*
    - ***Reference count*** *(count links to the cell, when ==0, reclaim)*
    - Generally mix of two schemes used.
- Dynamic type checking.
    - **tagged memories and special type-checking h/w**

The SECD Abstract Machine

Memory



The SECD Abstract Machine

Basic Data Structures

- Arbitrary s-expressions for computed data.
- List representing programs to be executed.
- Stack’s used by programs instructions.
- Value Lists containing arguments for uncompleted function applications.
- Closures to represent unprocessed function applications.

The SECD Abstract Machine

Machine Registers

- **S – Register** (Stack register)
  - Points to a list in memory that's treated as a conventional stack for built-in functions (+, -, etc)
  - Objects to be processed are pushed on by **cons**'ing a new cell on top of the current stack and **car** of this points to object's value.
  - S- register after such a push points to the new cell.
  - Unlike conventional stack, this does not overwrite original inputs.
  - Cells garbage collected later.

The SECD Abstract Machine

Machine Registers

- **<u>E – Register</u>** (Environment register)
  - Points to current value list of function arguments
    - The list is referenced by m/c when a value for the argument is needed.
    - List is augmented when a new environment for a function is created.
    - It's modified when a previously created closure is unpacked and the pointer from the closure's **cdr** replaces the contents of E-register.
  - Prior value list designated by E is not overwritten.

The SECD Abstract Machine

Machine Registers

- **C – Register** (Control register/pointer)
  - Acts as the program counter and points to the memory cell that designates through it’s **car** the next instruction to be executed.
  - The instructions are simple integers specifying desired operation.
  - Instructions do not have any sub-fields for registers etc. If additional information is required, it’s accessed through from the cells chained through the instruction cell’s **cdr**.
  - **“Increment of PC”** takes place by replacement of C registers contents by the contents of the last cell used by the instruction.
  - For return from completed applications, new function calls and branches, the C register is replaced by a pointer provided by some other part of the m/c.

The SECD Abstract Machine

Machine Registers

- **D – register** (Dump register)
  - Points to a list in memory called **“dump”**.
  - This data structure remembers the state of a function application when a new application in that function body is started.
  - That is done by appending onto dump the 3 new cells which record in their **cars** the value of registers **S**, **E**, and **C**.
  - When the application completes, popping the top of the dump restores those registers. This is very similar to **call-return** sequence in conventional m/c for procedure return and activation.

The SECD Abstract Machine

Basic Instruction Set

- Instruction can be classified into following 6 groups:
  1. Push object values onto the S stack.
  2. Perform built-in function applications on the S stack and return the result to that stack.
  3. Handle the if-then-else special form.
  4. Build, apply and return from closures representing non-recursive function applications.
  5. Extend the above to handle recursive functions.
  6. Handle I/O and machine control.

The CADR machine built at MIT (1984) closely resembles SECD with some non-trivial differences.

Case Study

Concert machine for MultiLISP (1985)

- **MultiLISP**
  - designed as an extension of SCHEME that permits the programmer to specify parallelism and then supports the parallelism in h/w “efficiently”.
- **SCHEME + new calls:**
  1. (PCALL F E1 E2 … En)
     - Permit parallel evaluation of arguments, then evaluate (F E1 E2 … En)
  2. (DELAY E)
     - Package E in closure.
  3. (TOUCH E)
     - Do not return until E evaluated.
  4. (FUTURE E)
     - Package E in a closure and permit eager evaluation
  5. (REPLACE-xxx E1 E2) [xxx is either CAR or CDR]
     - Replace xxx component of E1 by E2. (permits controlled modification to storage)
  6. (REPLACE-xxx-EQ E1 E2 E3)
     - Replace xxx of E1 by E2 iff xxx = E3. (TEST_AND_SET)

Case Study

Concert machine for MultiLISP (1985)

- Concert m/c at MIT – 24-way Motorola 68000 based shared memory multiprocessor.
- MultiLISP → ***MCODE*** (SECD-like ISA) → Interpreted by C interpreter (~ 3000 loc)
- Common gc heap distributed among all processor memories to hold all shared data.
- MCODE programs manage data structure called ***tasks*** that are accessed by 3pointers: *program pointer, stack pointer, and environment pointer.*

Case Study

Concert machine for MultiLISP (1985)

- *FUTURE* call creates a new task and leaves it accessible for any free processor. It's environment is that of it's parent expression at it's time creation.
- ***Task queue*** used to maintain schedulable tasks and ***unfair scheduling policy*** used to prevent task explosion.
- GC uses Banker's algorithm and spread over all processors with careful synchronization to avoid multiple processors trying to evacuate same object at the same time.

Questions?

Thanks for your patience …