Second-Generation Stack Computer Architecture And Self-Extensible Language

Introduction

- Discovered field by chance in 2000 (blame the Internet)
- Hobby project (simulations and assembly) until 2004
- Transformed into Independent Study thesis project
- Overview of current state of research
- Focus on programmer's view

Part 1: History And Arguments

Stack Computers: Origins

- First conceived in 1957 by Charles Hamblin at the University of New South Wales, Sydney.
- Derived from Jan Lukasiewicz's Polish Notation.
- Implemented as the GEORGE (General Order Generator) autocode system for the DEUCE computer.
- First hardware implementation of LIFO stack in 1963: English Electric Company's KDF9 computer.

Stack Computers: Origins (Part 2)

- Independently discovered in 1958 by Robert S. Barton (US).
- Implemented in the Burroughs B5000 (also in 1963).
- Better known
- Spawned a whole family of stack computers
- The First Generation

The First Generation: Features

- Multiple independent stacks in main memory
- Stacks are randomly accessible data structures
- Contained procedure activation records
- Evaluated expressions in Reverse Polish Notation
- Complex instructions sets trying to directly implement high-level languages (e.g.: PL/1, FORTRAN, ALGOL)
- Few hardware buffers (four or less typically)
- Supplanted in the 1980's by RISC and better compilers

Stack Computers: A New Hope

- Enter Charles H. (“Chuck”) Moore:
  - Creator of the stack-based FORTH language, circa 1970
  - Left Forth, Inc. in 1981 to pursue hardware implementations
    - NOVIX (1986), Sh-BOOM (1991), MuP21 (1994), F21 (1998), X18 (2001)
  - Currently CTO of Intelasys, still working on hardware
    - product launch expected April 3, 2006 at Microprocessor Summit
- Enter Prof. Philip Koopman, Carnegie-Mellon University
  - Documented salient stack designs in “*Stack Computers: The New Wave*”, 1989
  - The Second Generation

The Second Generation: Features

- Two or more stacks **separate from main memory**
- Stacks are **not addressable data structures**
- Expression evaluation and return addresses kept separate
- Simple instruction sets tailored for stack operations
- Still around, but low-profile (RTX-2010 in NASA probes)
- Strangely, missed by virtually all mainstream literature
  - Exception: Feldman & Retter's "*Computer Architecture*", 1993

Arguments and Defense

- Taken from Hennessy & Patterson's "*Computer Architecture: A Quantitative Approach*", 2nd edition
- Summary: Valid for First Generation, but not Second

Argument: Variables

More importantly, registers can be used to hold variables. When variables are allocated to registers, the memory traffic reduces, the program speeds up (since registers are faster than memory), and the code density improves (since a register can be named with fewer bits than a memory location).
[H&P, 2nd ed, pg 71]

- Manipulating the stack creates no memory traffic
- Stacks can be faster than registers since no addressing is required
- Lack of register addressing improves code density even more (no operands)
- Globals and constants are kept in main memory, or cached on stack for short sequences of related computations
- Ultimately no different than a register machine

Argument: Expression Evaluation

*Second, registers are easier for a compiler to use and can be used more effectively than other forms of internal storage. For example, on a register machine the expression (A*B)-(C*D)-(E*F) may be evaluated by doing the multiplications in any order, which may be more efficient due to the location of the operands or because of pipelining concerns (see Chapter 3). But on a stack machine the expression must be evaluated left to right, unless special operations or swaps of stack position are done.*

[H&P, 2nd ed, pg. 71]

- Less pipelining is required to keep a stack machine busy
- Location of operands is always the stack: no WAR, WAW dependencies
- However: always a RAW dependency between instructions
- Infix can be easily compiled to postfix
  - Dijkstra's “shunting yard” algorithm
- Stack swap operations equivalent to register-register move operations
- Stack are inverse registers!

Argument: Stacks Are Bad, Mmmm'kay?

1. *Performance is derived from fast registers, not the way they are used.*
2. *The stack organization is too limiting and requires many swap and copy operations.*
3. *The stack has a bottom, and when placed in slower memory there is a performance loss.*

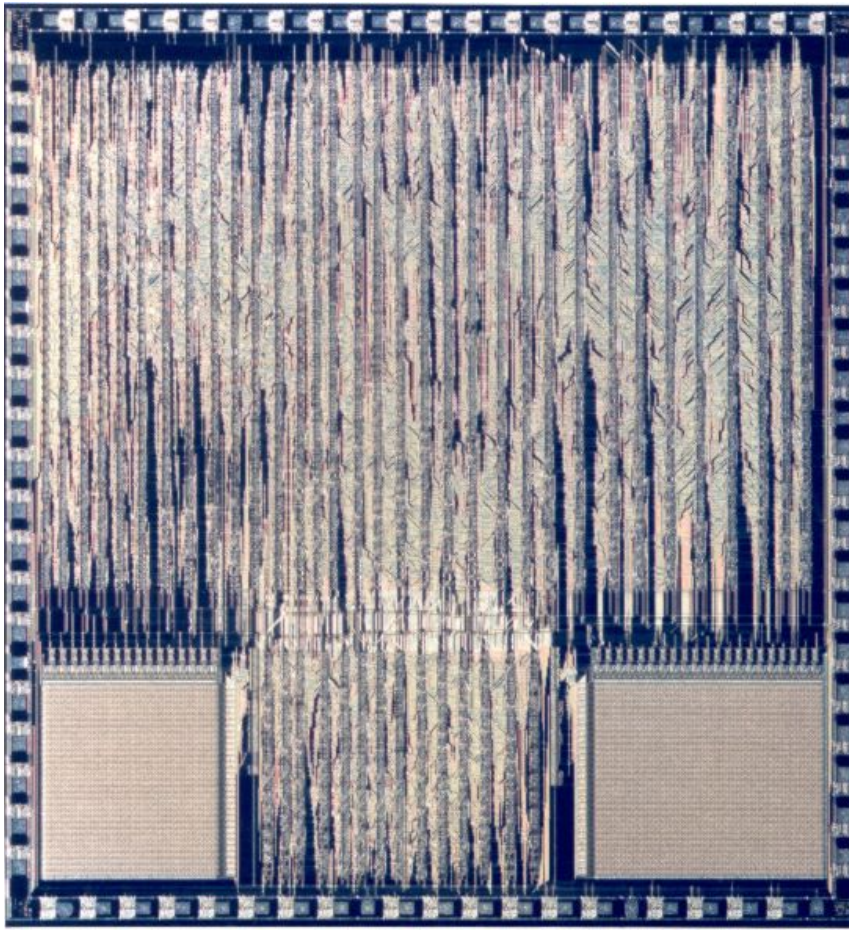
[H&P, 2nd ed, pg. 113]

(citing older sources)

1. Then it shouldn't matter that they're in a stack, and it's faster anyway.
2. Figure out sequence of operations at design or compile time. Swaps and copy operations can replace moves, loads and stores in some cases.
3. Strawman argument: all stacks have a bottom, and stacks 16-deep or more rarely require spilling to memory (< 1% of instructions).

Part 2: The Hardware

Example: Harris RTX-2000



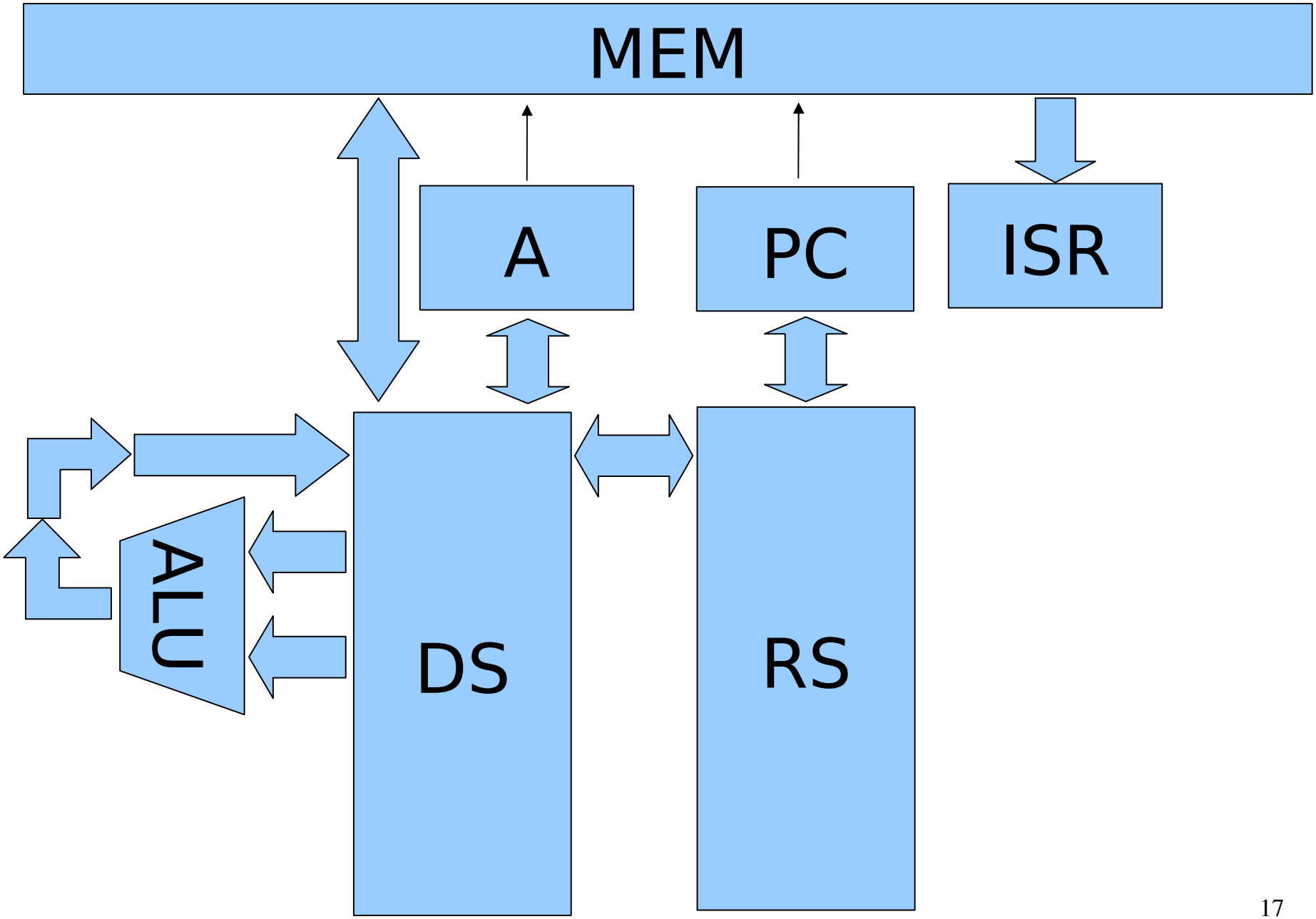
- 2.0u standard cell
- ~10MHz
- 16x16 multiplier
- 3 counters/timers
- ISR latency: 400ns!
- unencoded opcodes
- later Rad-Hard

The Hardware: Assumptions

- All registers/elements are word-wide (let's say 32 bits)
- Word addressing only (no bytes!)
- Flat memory model (no pages or segments)
- Signed integer arithmetic only (no overflow, no carry)
- No interrupts, no exceptions, memory-mapped I/O
- Zero-operand instruction set
  - Six 5-bit instructions per memory word (not VLIW!)
  - or one 32-bit integer

The Hardware: Major Blocks

- Data Stack (DS)
- Return Stack (RS)
- Address Register (A)
- ALU
- Instruction Shift Register (ISR)
- Program Counter (PC)


MEM
A
PC
ISR
ALU
DS
RS

Instruction Set

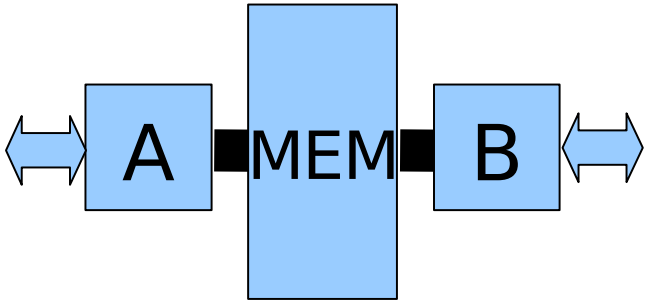
- Data Stack
  - LIT, XOR, AND, NOT, 2*, 2/, +, +*, DUP, DROP, OVER
- Return Stack
  - CALL, RET, JMP, JMP0, JMP+, R@+, R!+, >R, R>
- Address Register
  - >A, A>, A@, A!, A@+, A!+
- Other
  - NOP, UNDEF (4 left), PC@ (free!)

So What's It Good For?

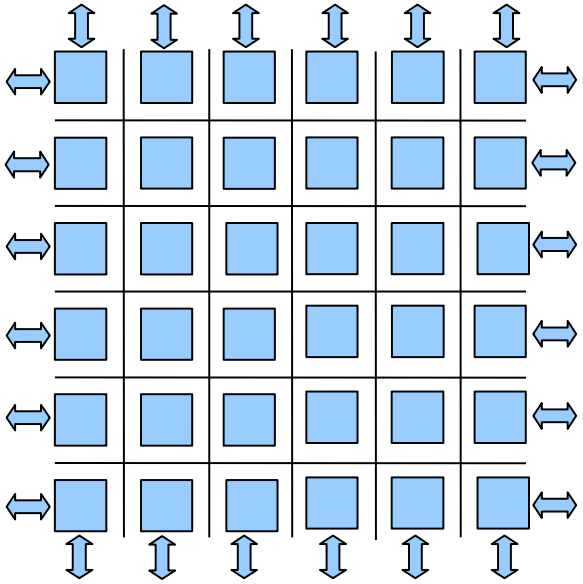
- Fast procedure calls
  - fast interrupts
- Compact code
- Reduced system complexity
  - Shorter pipeline
  - Simpler compilation
- Consistent *instantaneous* performance
  - Hard Real-Time
- For small systems: more !/$

The Future

- Pairs



- Sea-of-Processors



Asynchronous!

Part 3: The Software

The Software: Time

- C: Edit-Compile-Execute
- Lisp: Read-Eval-Print (REPL)

- Compile Time
- Run Time

The Software: Kernel

- Structures:
  - Name & Code Dictionaries, Input Buffer, Counted Strings
- Variables:
  - HERE, HERE_NEXT, THERE, INPUT, SLOT, NAME_END
- Procedures:
  - (CALL), (RET), (JMP), (JMP0), (JMP+), (LIT)
  - (DROP), (+), (A@+), etc...
  - NUMI, SCAN, DEFN, LOOK, EXECUTE, NXEC
- How big? ~800 32-bit words (or less!)

The Software: Raw Stuff

- SCAN : DEFN

SCAN SCAN LOOK POP_STRING (CALL)

SCAN DEFN LOOK POP_STRING (CALL) (RET)

  - call SCAN call DEFN RET

- : I

SCAN SCAN LOOK POP_STRING (CALL)

SCAN LOOK LOOK POP_STRING (CALL)

SCAN POP_STRING LOOK POP_STRING (CALL) (RET)

  - call SCAN call LOOK call POP_STRING RET

The Software: Medium-Rare

- : $c l LOOK (CALL) l POP_STRING (CALL) l (CALL) (CALL) (RET)
  - call LOOK call POP_STRING call (CALL) RET

- : c SCAN SCAN $c SCAN $c $c RET
  - call SCAN call $c RET

- : n c SCAN c NUMI (RET)
  - call SCAN call NUMI RET

Macros: if...else

- : abs (DUP) if- c negate ; else ;

- : if- n# 0 c (JMP+) l# HERE_NEXT (@) (N-) 1 ;
- : else (>R) c NEW_WORD (R>) (!) ;

- DUP JMP+ call negate RET RET

Macros: Higher-Order Functions

Source

```
: mapgen
c (DUP) c (>R)
l# STRING_TAIL c (CALL) c
   (R>)
n 1 # c # c (+)
c NEW_WORD
c (DUP) c l c (CALL) (>R) c #
   (R>) c (+)
c (OVER) c (OVER) c (XOR)
c if (>R) c (JMP) (R>)
c else c (DROP) c (DROP) c ; ;
```

Object

```
(DUP) (>R)
[LIT STRING_TAIL] (CALL) (R>)
[LIT 1] (LIT) (+)
NEW_WORD
(DUP) l (CALL) >R (LIT) R> (+)
(OVER) (OVER) (XOR)
if >R (JMP) R>
else (DROP) (DROP) (RET) RET
```

Macros: Higher-Order Functions (2)

Object

(DUP) (>R)

[LIT STRING_TAIL] (CALL) (R>)

[LIT 1] (LIT) (+)

NEW_WORD

(DUP) l (CALL) >R (LIT) R> (+)

(OVER) (OVER) (XOR)

if >R (JMP) R>

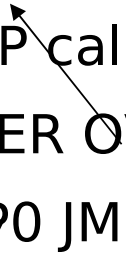
else (DROP) (DROP) (RET) RET

- : cipher n 2 mapgen encoder
- Compiles:

DUP >R

call STRING_TAIL R>

[LIT 1] +



DUP call encoder [LIT 2] +

OVER OVER XOR

JMP0 JMP

DROP DROP RET

- SCAN BlahBlahBlahBlahBlah l INPUT @ cipher

The Software: In Progress

- Source code:
  - Extensions: 6671 bytes
  - Simple virtual machine: 3903 bytes
  - Metacompiler: 6641 bytes
  - Simple RPC: 1507 bytes
- Total binary size, incl. Kernel: < 6000 kwords (32-bit)

Thank You!

Questions?