

# Complex Instruction Set Evolution in the Sixties and Seventies: *Stack and GPR Architectures*

**Krste Asanovic**
**Laboratory for Computer Science**
**M.I.T.**

**http://www.csg.lcs.mit.edu/6.823**



## The Sixties

- ***Hardware costs started dropping***
  - **memories beyond 32K words seemed likely**
  - **separate I/O processors**
  - **large register files**
- ***Systems software development became essential***
  - **Operating Systems**
  - **I/O facilities**
- ***Separation of Programming Model from implementation become essential***
  - **family of computers**



# *The Burrough's B5000:*

## *An ALGOL Machine, Robert Barton, 1960*

***Machine implementation can be completely hidden if the programmer is provided only a high-level language interface.***

***Stack machine* organization because stacks are convenient for:**

1. **expression evaluation;**
2. **subroutine calls, recursion, nested interrupts;**
3. **accessing variables in block-structured languages.**

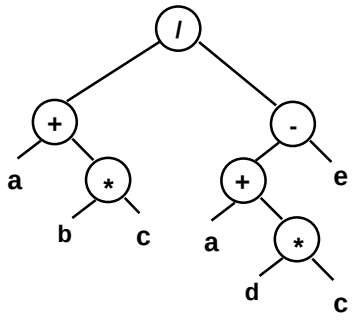
**B6700, a later model, had many more innovative features**

- ***tagged data***
- ***virtual memory***
- ***multiple processors and memories***



# Evaluation of Expressions

**(a + b * c) / (a + d * c - e)**

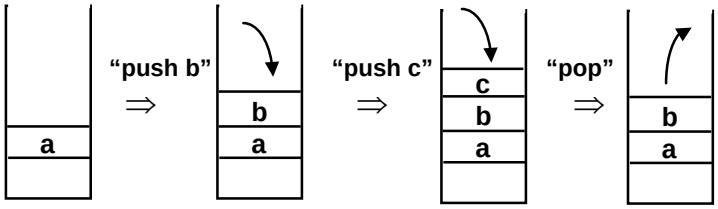



**Reverse Polish**

**a b c * + a d c * + e - /**



# A Stack Machine

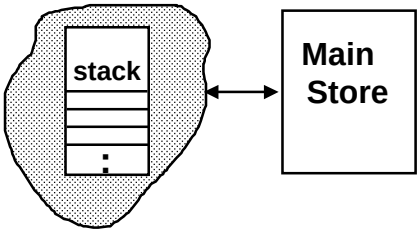



***A Stack machine has a stack as a part of the processor state.***

**typical operations:**

- **push**
- **pop**
- **+**
- ***...**

***Instructions like + implicitly specify the top 2 elements of the stack as operands.***






# Hardware organization of the stack

- **Stack is part of the processor state**
  - ⇒ ***stack must be bounded and small***
    - ≈ **number of Registers and *not* the size of main memory**
- **Conceptually stack is unbounded**
  - ⇒ ***a part of the stack is included in the processor state; the rest is kept in the main memory***



# Stack Operations and Implicit Memory References

**Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.**

**Each *push* operation ⇒ 1 memory reference.**

**pop operation ⇒ 1 memory reference.**

***No Good!***

**Better performance can be gotten by keeping the top N elements in registers and by making memory references only when register stack overflows or underflows.**

***Issue - when to Load/Unload registers ?***



# Stack Size and Memory References

**a b c * + a d c * + e - /**

| *program* | *stack* | *memory refs (size = 2)* |
|---|---|---|
| push a | R0 | a |
| push b | R0 R1 | b |
| push c | R0 R1 R2 | c, ss(a) |
| * | R0 R1 | sf(a) |
| + | R0 | |
| push a | R0 R1 | a |
| push d | R0 R1 R2 | d, ss |
| push c | R0 R1 R2 R3 | c, ss |
| * | R0 R1 R2 | sf |
| + | R0 R1 | sf |
| push e | R0 R1 R2 | e,ss |
| - | R0 R1 | sf |
| / | R0 | *4 stores, 4 fetches (implicit)* |



# Stack Size and Expression Evaluation

**a b c * + a d c * + e - /**

*a and c are "loaded" twice*

⇒

*not the best use of registers!*

| | |
|---|---|
| **push a** | **R0** |
| **push b** | **R0 R1** |
| **push c** | **R0 R1 R2** |
| ***** | **R0 R1** |
| **+** | **R0** |
| **push a** | **R0 R1** |
| **push d** | **R0 R1 R2** |
| **push c** | **R0 R1 R2 R3** |
| ***** | **R0 R1 R2** |
| **+** | **R0 R1** |
| **push e** | **R0 R1 R2** |
| **-** | **R0 R1** |
| **/** | **R0** |



# Register Usage in a GPR Machine

**(a + b * c)/(a + (d * c) - e)**

| | | |
|---|---|---|
| **Load** | **R0** | **a** |
| **Load** | **R1** | **c** |
| **Load** | **R2** | **b** |
| **Mul** | **R2** | **R1** |
| **Add** | **R2** | **R0** |
| **Load** | **R3** | **d** |
| **Mul** | **R3** | **R1** |
| **Add** | **R3** | **R0** |
| **Load** | **R0** | **e** |
| **Sub** | **R3** | **R0** |
| **Div** | **R2** | **R3** |

*More control over register usage since registers can be named explicitly*

**Load Ri m**
**Load Ri (Rj)**
**Load Ri (Rj) (Rk)**

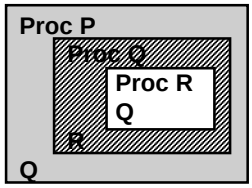
⇒

*- eliminates unnecessary Loads and Stores*
*- fewer Registers*

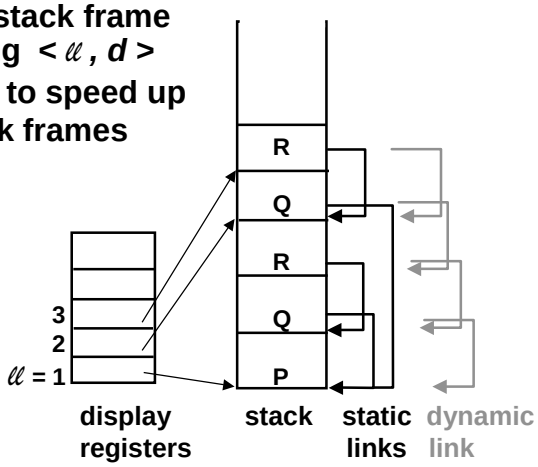
*but instructions may be longer!*

# Procedure Calls

- Storage for procedure calls also follows a stack discipline
- However, there is a need to access variables beyond the current stack frame
  - lexical addressing $< \ell , d >$
  - display registers to speed up accesses to stack frames




*automatic loading of display registers?*

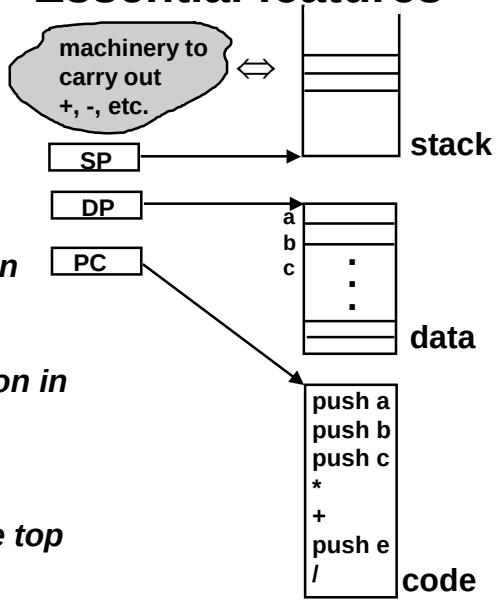





# Stack Machines -- Essential features

In addition to push, pop, + etc., the instruction set must provide the capability to

- *refer to any element in the data area*
- *jump to any instruction in the code area*
- *move any element in the stack frame to the top*





# Stack versus GPR Organization

*Amdahl, Blaauw and Brooks, 1964*

1. **The performance advantage of push down stack organization is derived from the presence of fast registers and not the way they are used.**
2. **"Surfacing" of data in stack which are "profitable" is approximately 50% because of constants and common sub-expressions.**
3. **Advantage of instruction density because of implicit addresses is equaled if short addresses to specify registers are allowed.**
4. **Management of finite depth stack causes complexity.**
5. **Recursive subroutine advantage can be realized only with the help of an independent stack for addressing.**
6. **Fitting variable length fields into fixed width word is awkward.**



# Stack Machines (Mostly) Died by 1980

1. **Stack programs are not smaller if short (Register) addresses are permitted.**

2. **Modern compilers can manage fast register space better than the stack discipline.**

3. **Lexical addressing is a useful abstract model for compilers but hardware support for it (i.e. display) is not necessary.**

***GPR's and caches are better than stack and displays***

***Early language-directed architectures often did not take into account the role of compilers!***

**B5000, B6700, HP 3000, ICL 2900, Symbolics 3600**



# Stacks post-1980

- **Inmos Transputers (1985-2000)**
  - **Designed to support many parallel processes in Occam language**
  - **Inmos T800 was world's fastest microprocessor in late 80's**
  - **Fixed-height stack design simplified implementation**
  - **Stack trashed on context swap (fast context switches)**
- **Forth machines**
  - **Direct support for Forth execution in small embedded real-time environments**
  - **Several manufacturers (Rockwell, Patriot Scientific)**
- **Intel x87 floating-point unit**
  - **Severely broken stack model for FP arithmetic**
  - **Deprecated in Pentium-4 in exchange for SSE2 FP register arch.**
- **Java Virtual Machine**
  - **Designed for software emulation not direct hardware execution**
  - **Sun PicoJava implementation + others**



# IBM 360:
# *A General-Purpose Register Machine*

- ***Processor State***

  **16 General-Purpose 32-bit Registers**

  ***- may be used as index and base registers***

  ***- Register 0 has some special properties***

  **4 Floating Point 64-bit Registers**

  **A Program Status Word (PSW)**

  ***PC, Condition codes, Control flags***

- ***A 32-bit machine with 24-bit addresses***

  ***No instruction contains a 24-bit address !***

- ***Data Formats***

  **8-bit bytes, 16-bit half-words, 32-bit words,**

  **64-bit doublewords**



# IBM 360: Implementations

| | *Model 30 . . .* | *Model 70* |
|---|---|---|
| ***Storage*** | **8K - 64 KB** | **256K - 512 KB** |
| ***Datapath*** | **8-bit** | **64-bit** |
| ***Circuit Delay*** | **30 nsec/level** | **5 nsec/level** |
| ***Local Store*** | **Main Store** | **Transistor Registers** |
| ***Control Store*** | **Read only 1μsec** | **Conventional circuits** |

***IBM 360 instruction set architecture completely hid the underlying technological differences between various models.***

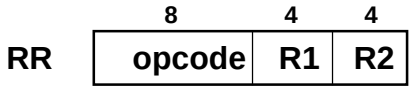


# IBM 360: Precise Interrupts

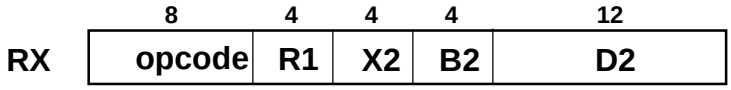
- **IBM 360 ISA (Instruction Set Architecture) preserves sequential execution model**
- **Programmers view of machine was that each instruction either completed or signaled a fault before next instruction began execution**
- **Exception/interrupt behavior constant across family of implementations**



# *IBM 360:* Some Addressing Modes




R1← (R1) op (R2)




R1 ← (R1) op M[(X2) + (B2) + D2]
a 24-bit address is formed by adding the 12-bit displacement (D) to a base register (B) and an Index register (X), if desired

***The most common formats for arithmetic & logic instructions, as well as Load and Store instructions***



# *IBM 360:* Branches & Condition Codes

- Arithmetic and logic instructions set *condition codes*
  - equal to zero
  - greater than zero
  - overflow
  - carry...
- I/O instructions also set condition codes - *channel busy*
- All conditional branch instructions are based on testing these condition code registers (CC's)

RX and RR formats

| | |
|---|---|
| BC_ | branch conditionally |
| BAL_ | branch and link, i.e., R15 ← (PC)+1 *for subroutine calls* |

⇒ CC's must be part of the PSW

# *IBM 360:* Character String Operations

| 8 | 8 | 4 | 12 | 4 | 12 |
|---|---|---|---|---|---|
| opcode | length | B1 | D1 | B2 | D2 |

**SS format: store to store instructions**

**M[(B1) + D1] ← M[(B1) + D1] op M[(B2) + D2]**
**iterate "length" times**

***most operations on decimal and character strings use this format***

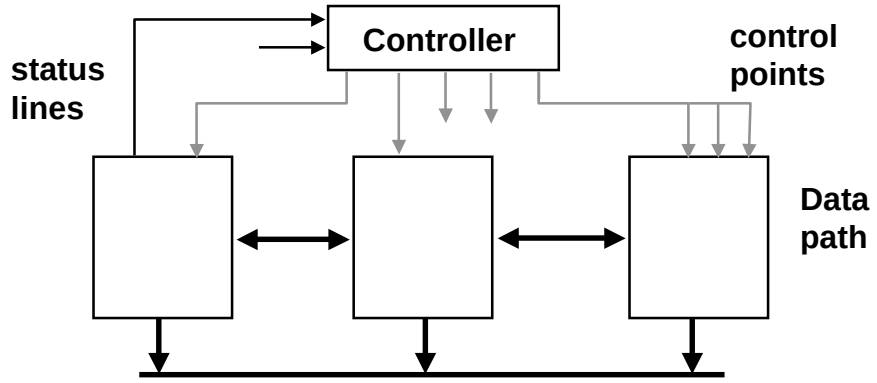
**MVC move characters**
**MP multiply two packed decimal strings**
**CLC compare two character strings**
**...**

***a lot of memory operations per instruction complicates exception & interrupt handling***



# Microarchitecture

***Implementation of an ISA***




***Structure:*** **How components are connected.**
***Static***

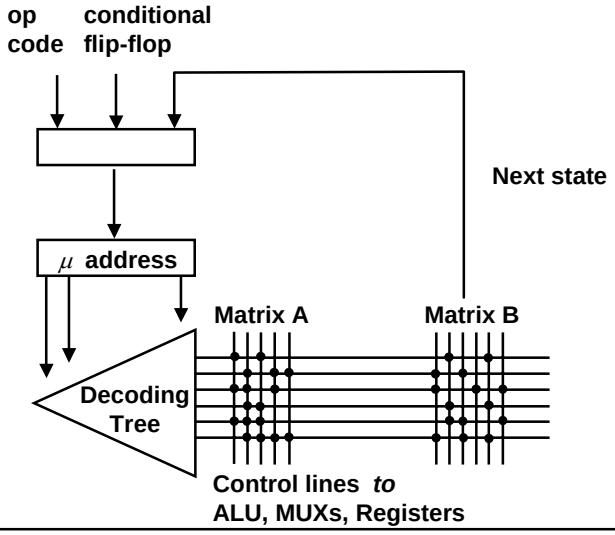
***Sequencing:*** **How data moves between components**
***Dynamic***



# Microcontrol Unit

***Maurice Wilkes, 1954***

***Embed the state table in a memory array***






# Seventies' General-Purpose Architectures

***(minicomputers and mainframes for business processing)***

## Characteristics:

- **Complex Instruction Sets:**
- **Microcoded Control**
- **Very Little Instruction Pipelining**
- **Caches**
- **Virtual Memory**

**(IBM 360->370, DEC PDP-11->VAX)**



# Software in the Seventies

- Most programming was done in *high-level languages*
  Fortran, COBOL, Basic, Lisp, Pascal, Simula, C, ...
- *Time-sharing* and *interactive computing* came into widespread use *(screens* versus *punched cards)*
- Most companies used ***proprietary*** Operating Systems
  IBM OS370, MVS, ...
  DEC VMS, ...
- Unix, a ***portable*** OS, was beginning to be used on minicomputers in universities
- Internet (e-mail, FTP, TCP/IP,...) came into existence



# Sources of Complexity in ISAs

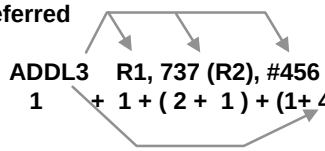
- Addressing modes
- Variable-length fields
- Macro-instructions
  - loop control
  - function call
- Regularity
  - most addressing mode for each operand
  - most operators for each data type

***Microprogramming made it possible to implement large and complex instruction sets at a reasonable cost***



# VAX: A Complex Instruction Set

| *Addressing Mode* | *Syntax* | *Length in bytes* |
|---|---|---|
| Literal | #value | 1 (6-bit signed value) |
| Immediate | #value | 1 + \|immediate\| |
| Register | Rn | 1 |
| Register deferred | (Rn) | 1 |
| Byte/word/long displacement | disp(Rn) | 1 + \|displacement\| |
| Byte/word/long displacement deferred | @disp(Rn) | 1 + \|displacement\| |
| Scaled (Indexed) | base mode (Rx) | 1 + \|base addressing mode\| |
| Autoincrement | (Rn)+ | 1 |
| Autodecrement | -(Rn) | 1 |
| Autoincrement deferred | @(Rn)+ | 1 |



ADDL3 R1, 737 (R2), #456 ≡ R1 <-- M[(R2)+737] + 456

1 + 1 + ( 2 + 1 ) + (1+ 4) = 10 bytes !



# VAX Extremes

## Long instructions, e.g.,

```
emodh #1.2334, 70000(r1)[r2],
      #2.3456, 80000(r1)[r2],
      90000(r1)[r2]
```

- 54 byte encoding (rumors of even longer instruction sequences...)

## Memory referencing behavior

- In worst case, a single instruction could require at least 41 different memory pages to be resident in memory to complete execution
- Doesn't include string instructions which were designed to be interruptable