

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

**Krste Asanovic**
**Laboratory for Computer Science**
**Massachusetts Institute of Technology**



# 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:**

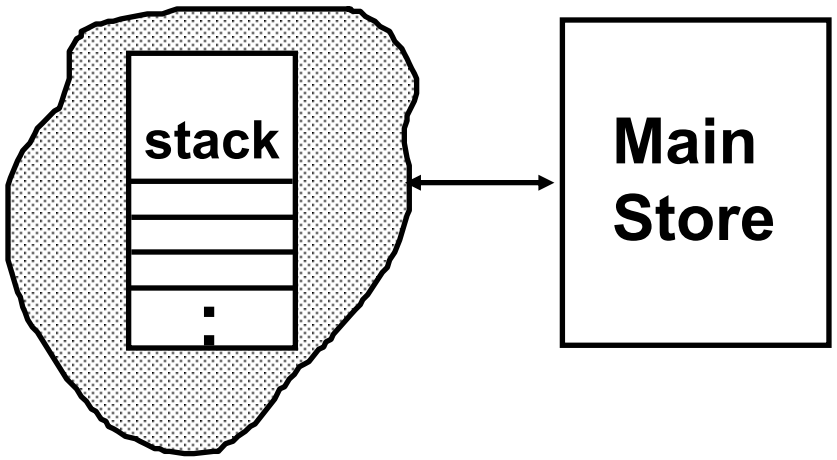
1. **expression evaluation;**
2. **subroutine calls, recursion, nested interrupts;**
3. **accessing variables in block-structured languages.**

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

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

# A Stack Machine

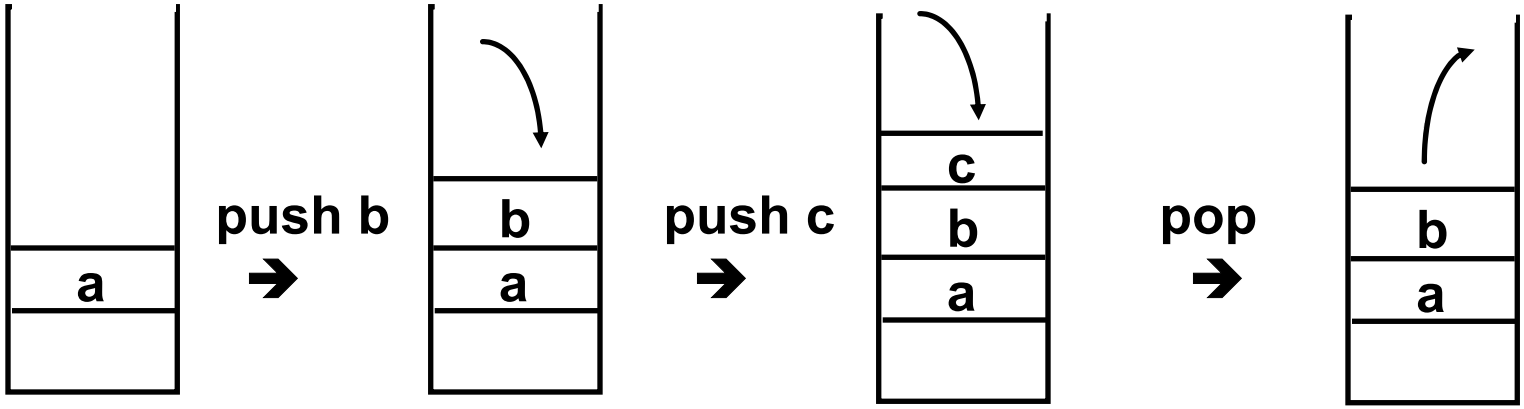
***A Stack machine has a stack as a part of the processor state***




**typical operations:**

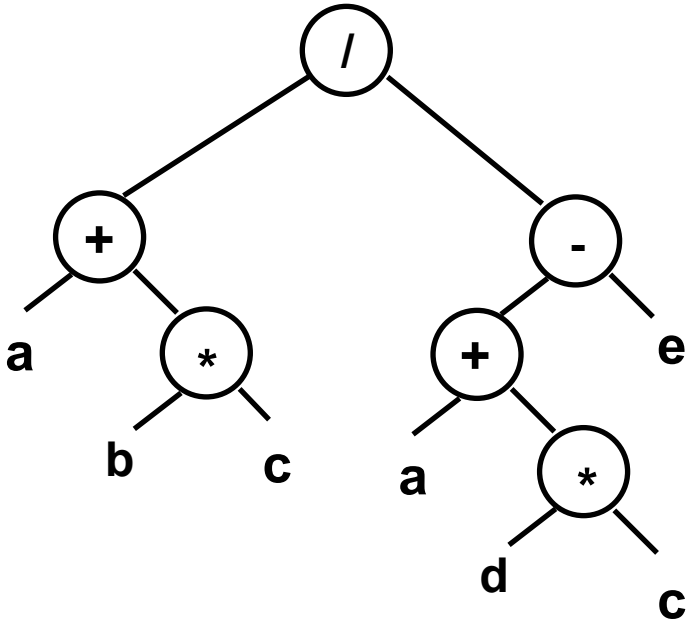
**push**
**pop**
**+**
***...**

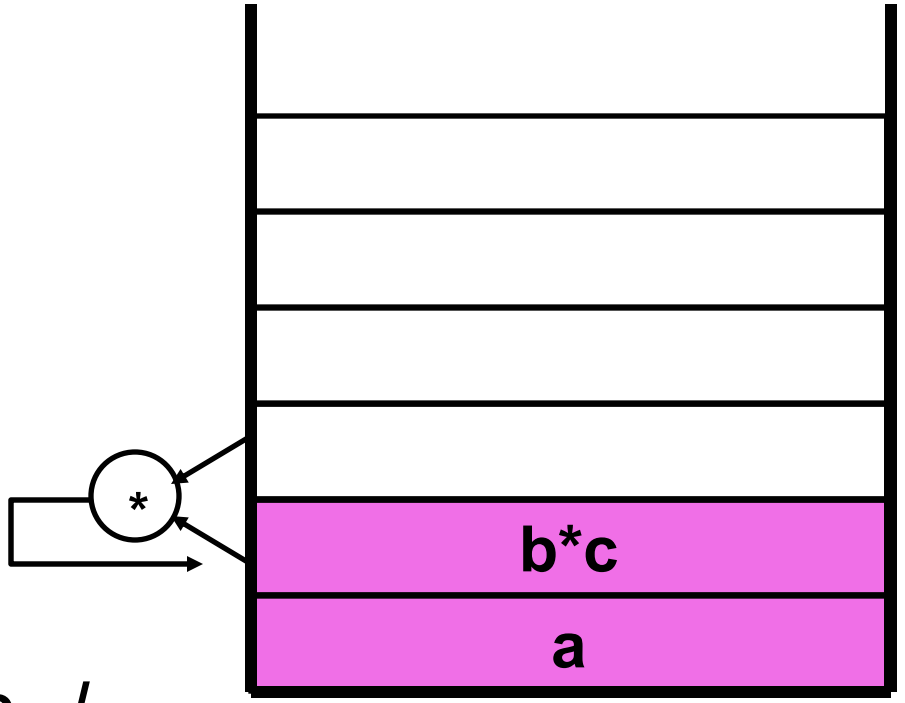
***Instructions like + implicitly specify the top 2 elements of the stack as operands.***



# Evaluation of Expressions

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




Evaluation Stack

Reverse Polish

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

Push a

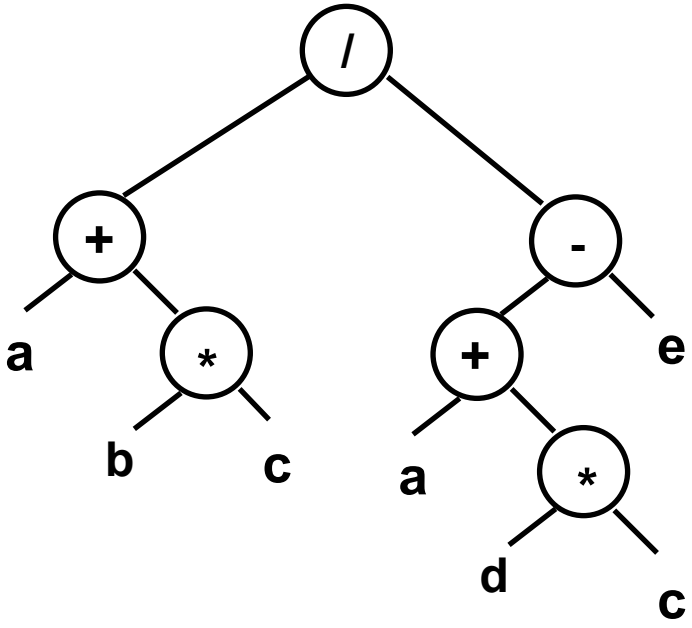
Push b

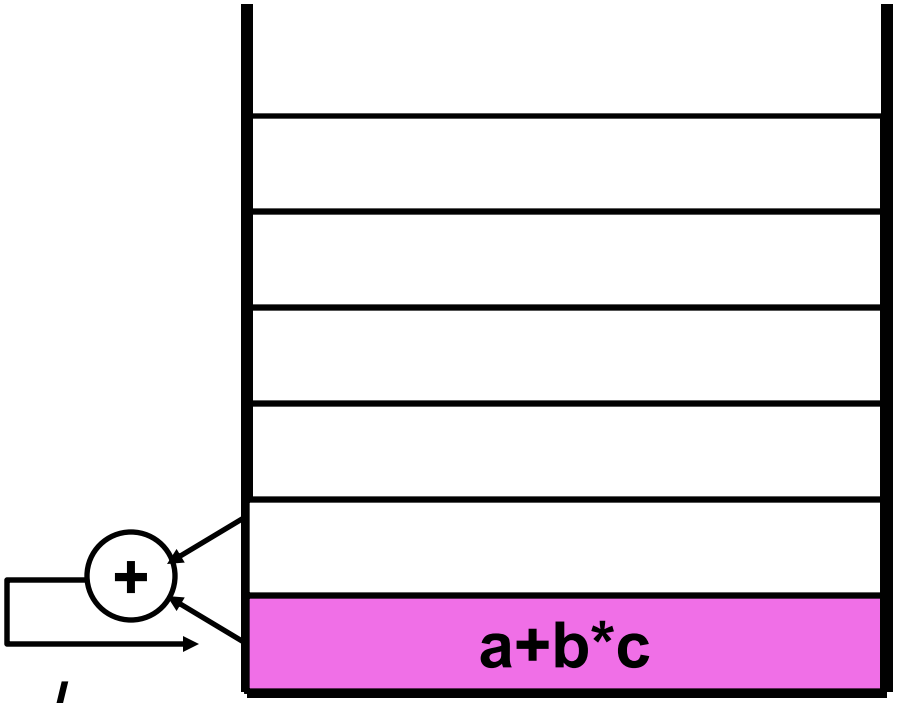
push c

multiply

# Evaluation of Expressions

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




Evaluation Stack

Reverse Polish

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

add



# Hardware Organization of the Stack

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



# Stack Size and Memory References

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

| *Program* | *Stack (size = 2)* | | *Memory Refs* |
|---|---|---|---|
| push a | a | | a |
| push b | a, | b | b |
| push c | b, | c | c, ss(a) |
| * | a, | b*c | sf(a) |
| + | a+b*c | | |
| push a | a+b*c, | a | a |
| push d | a, | d | d, ss(a+b*c) |
| push c | d, | c | c, ss(a) |
| * | a, | d*c | sf(a) |
| + | a+b*c, | a+d*c | sf(a+b*c) |
| push e | a+d*c, | e | e, ss(a+b*c) |
| - | a+b*c, | a+d*c-e | sf(a+b*c) |
| / | (a+b*c)/(a+d*c-e) | | |

***4 stores, 4 fetches (implicit)***



# Stack Operations and Implicit Memory References

**When just the top 2 elements of the stack are kept in registers and the rest is kept in the memory:**

**Each *push* operation $\Rightarrow$ 1 memory reference.**

***pop* operation $\Rightarrow$ 1 memory reference.**

***No Good!***

**Better performance from 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 ?***

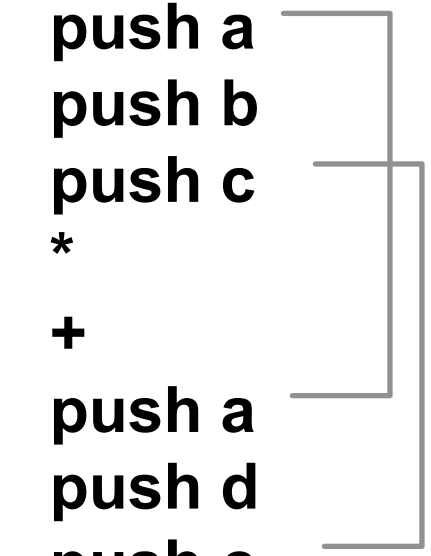
# Stack Size and Expression Evaluation

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

*a and c are "loaded" twice*

$\Rightarrow$

*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)
```

$\Rightarrow$

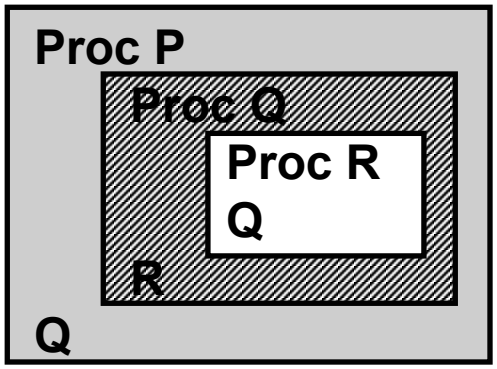
***- eliminates unnecessary Loads and Stores***

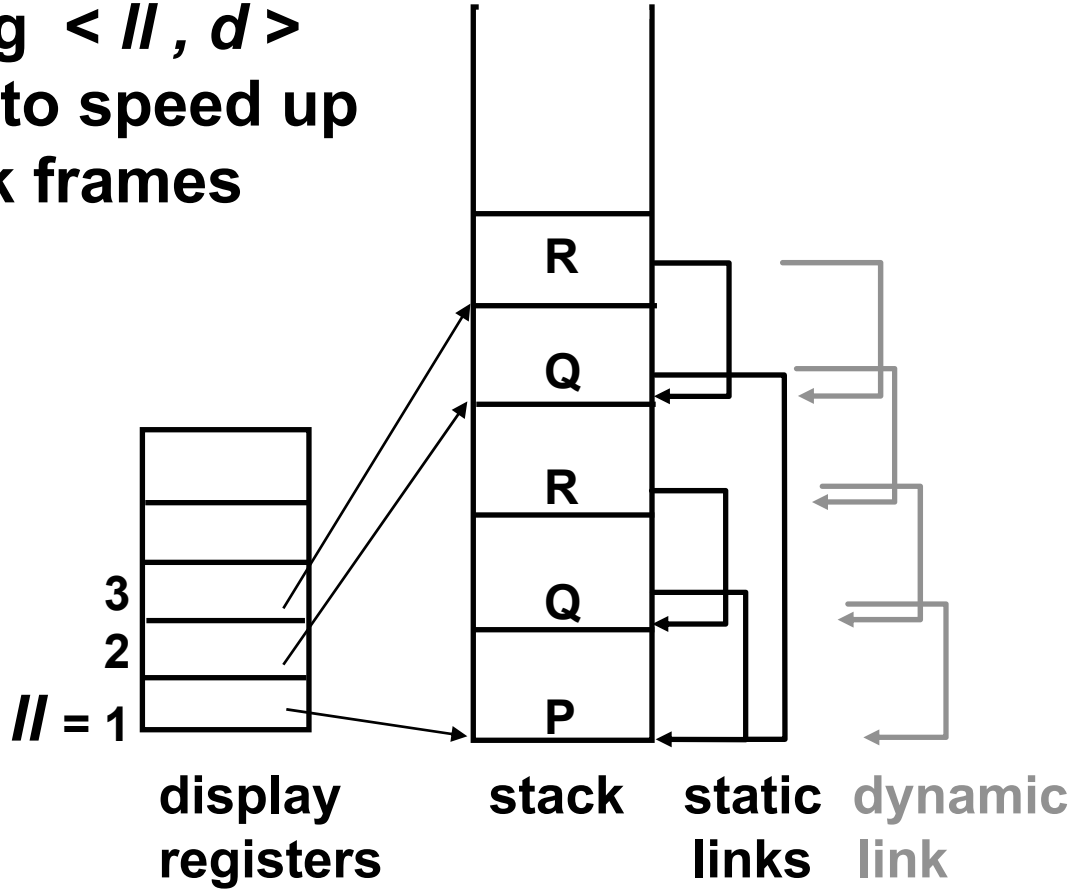
***- fewer Registers***

***but instructions may be longer!***

# B5000 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 < *ll* , *d* >
  - display registers to speed up accesses to stack frames



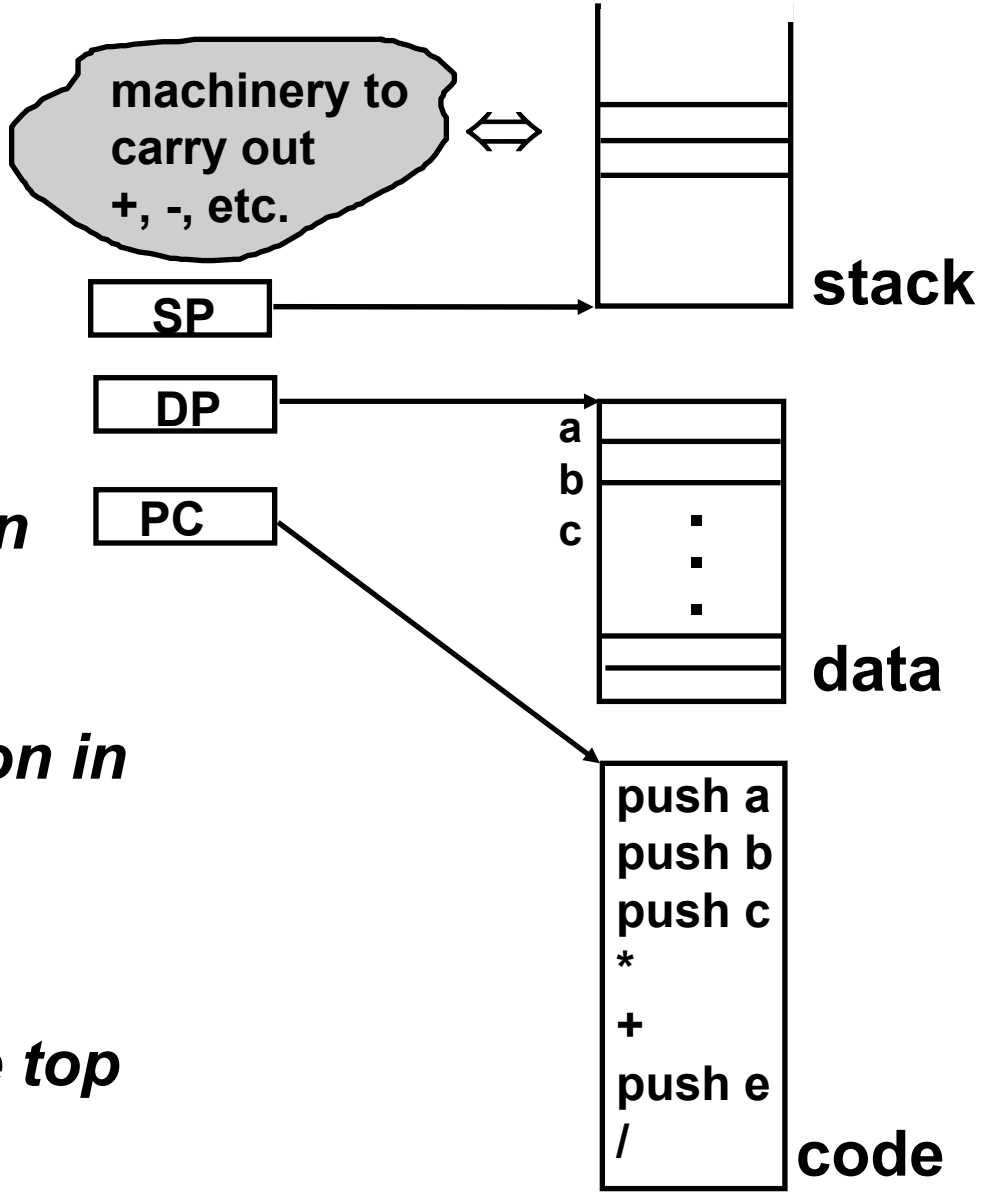




*automatic loading of display registers?*

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



# 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 |

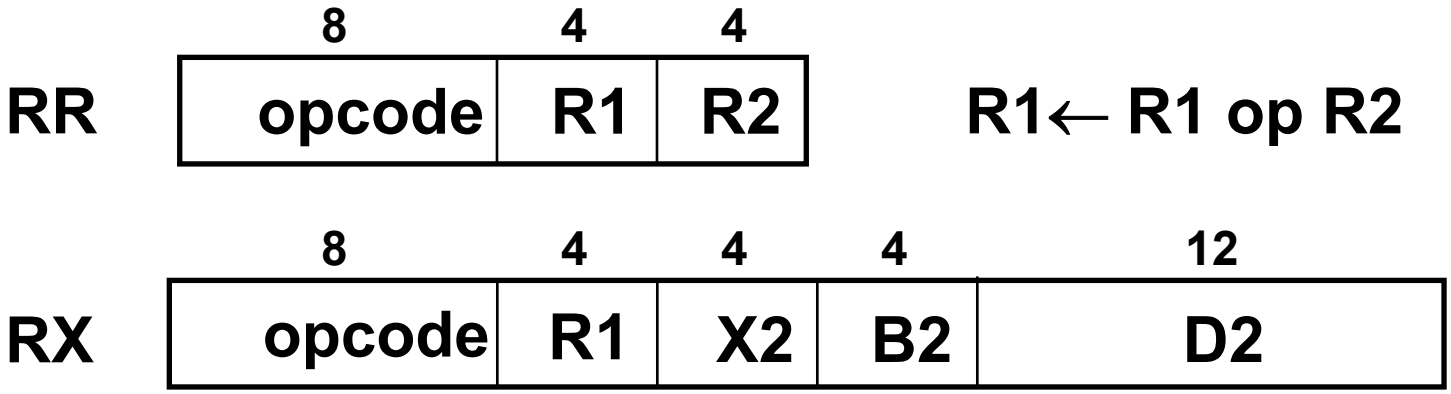
***IBM 360 instruction set architecture completely hid the underlying technological differences between various models.***



# 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 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)

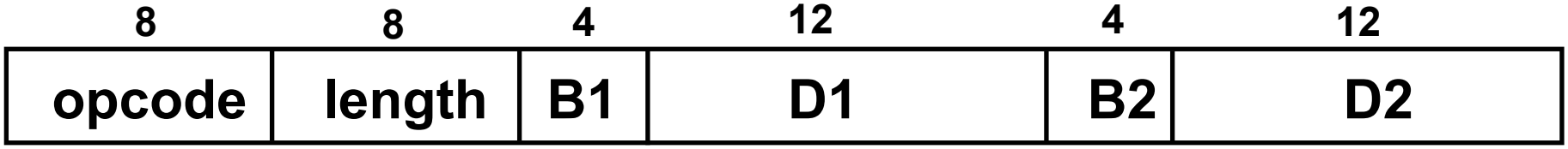
RX and RR formats

BC_ branch conditionally

BAL_ branch and link, i.e., R15 $\leftarrow$ PC + 1
*for subroutine calls*

$\Rightarrow$ CC's must be part of the PSW

# IBM 360: Character String Operations




**SS format: store to store instructions**

**M[B1 + D1] ← M[B1 + D1] op M[B2 + D2]**

**iterate "length" times**

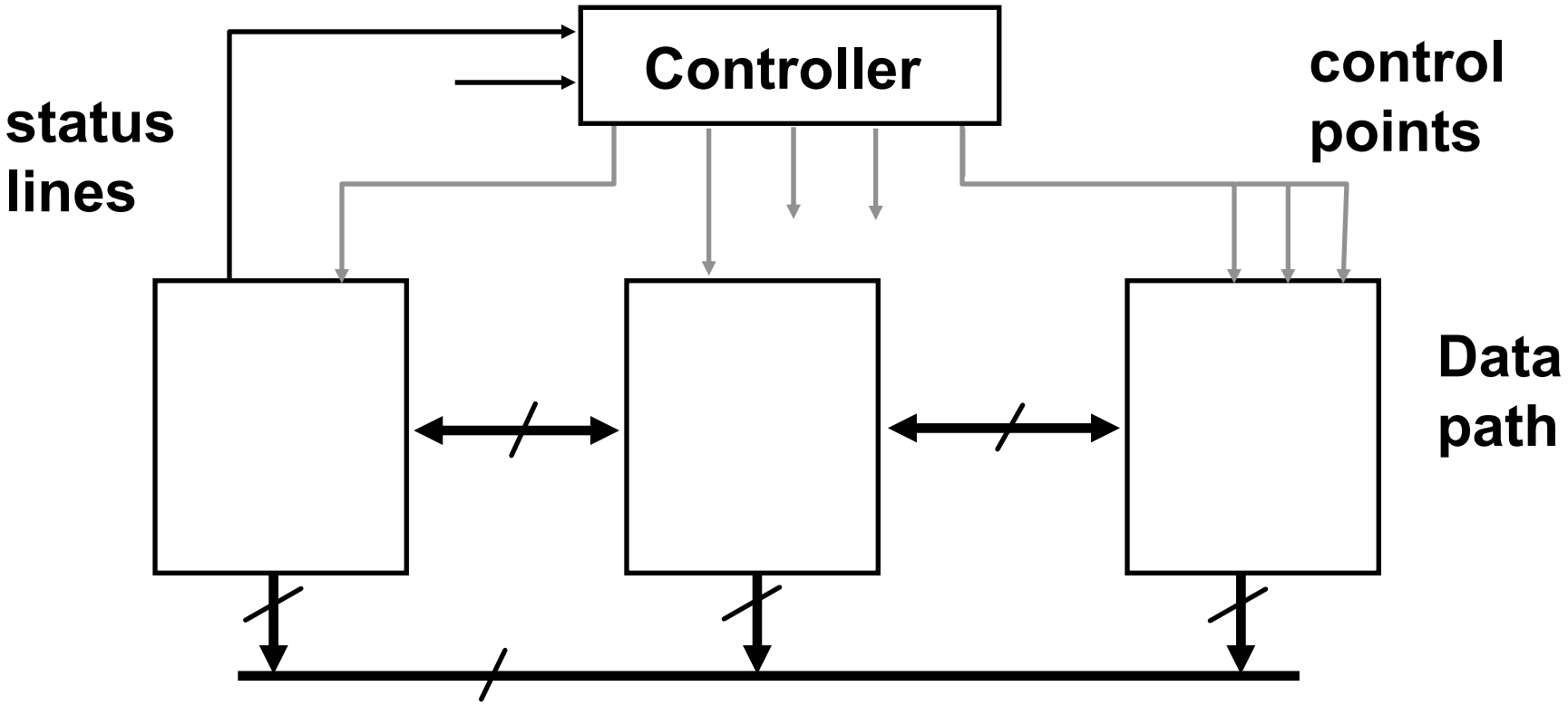
***most operations on decimal and character strings use this format***

**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***



# The DLX ISA

## *Processor State*

- 32 32-bit GPRs, R0 always contains a 0
- 32 single precision FPRs, may also be viewed as
  - 16 double precision FPRs
- FP status register, used for FP compares & exceptions
- PC, the program counter
- some other special registers

## *Data types*

- 8-bit byte, 2-byte half word
- 32-bit word for integers
- 32-bit word for single precision floating point
- 64-bit word for double precision floating point

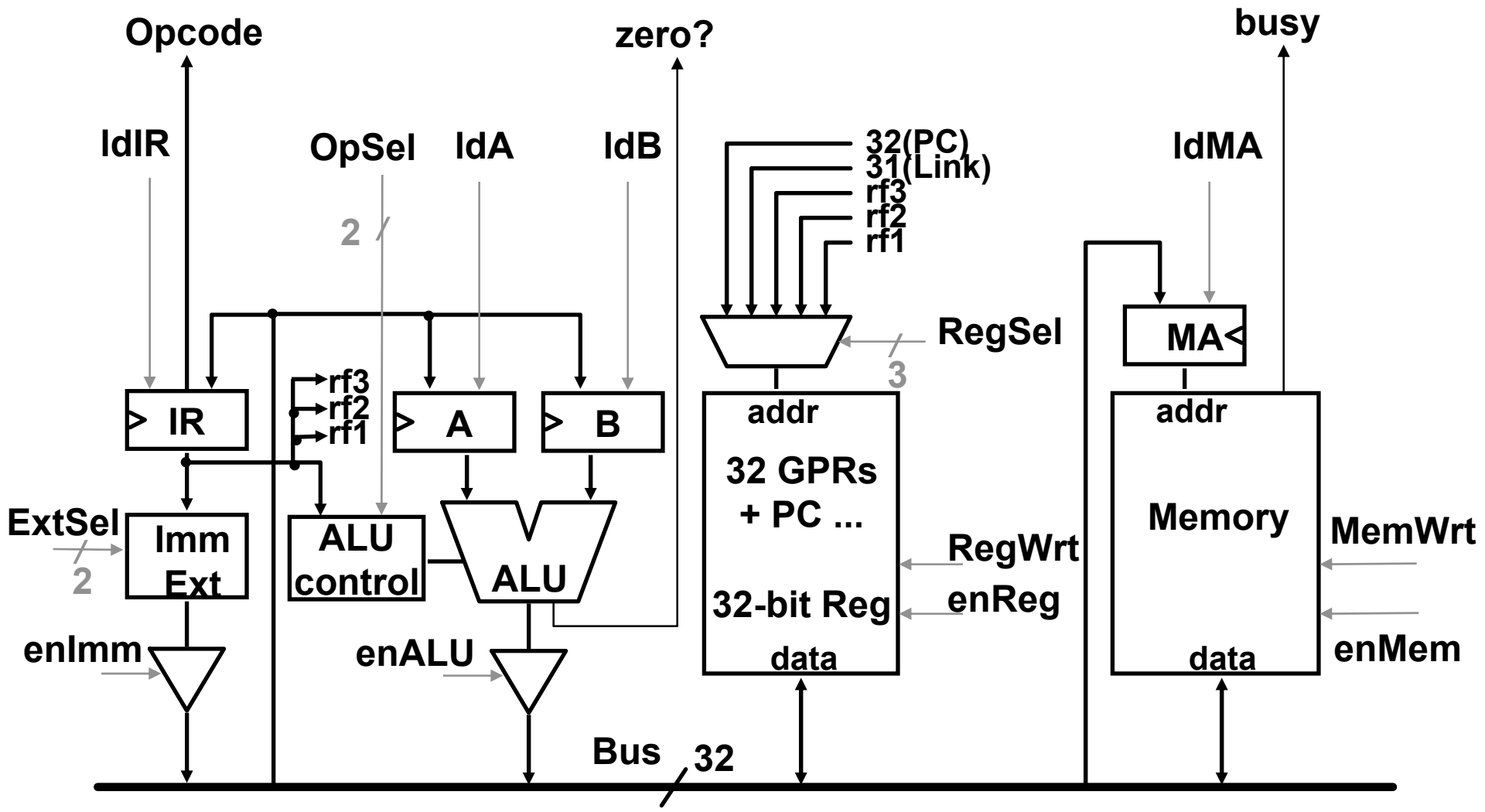
## *Load/Store style instruction set*

- data addressing modes- immediate & indexed
- branch addressing modes- PC relative & register indirect
- Byte addressable memory- big-endian mode

## *All instructions are 32 bits*

*(See Chapter 2, H&P for full description)*

# A Bus-based Datapath for DLX

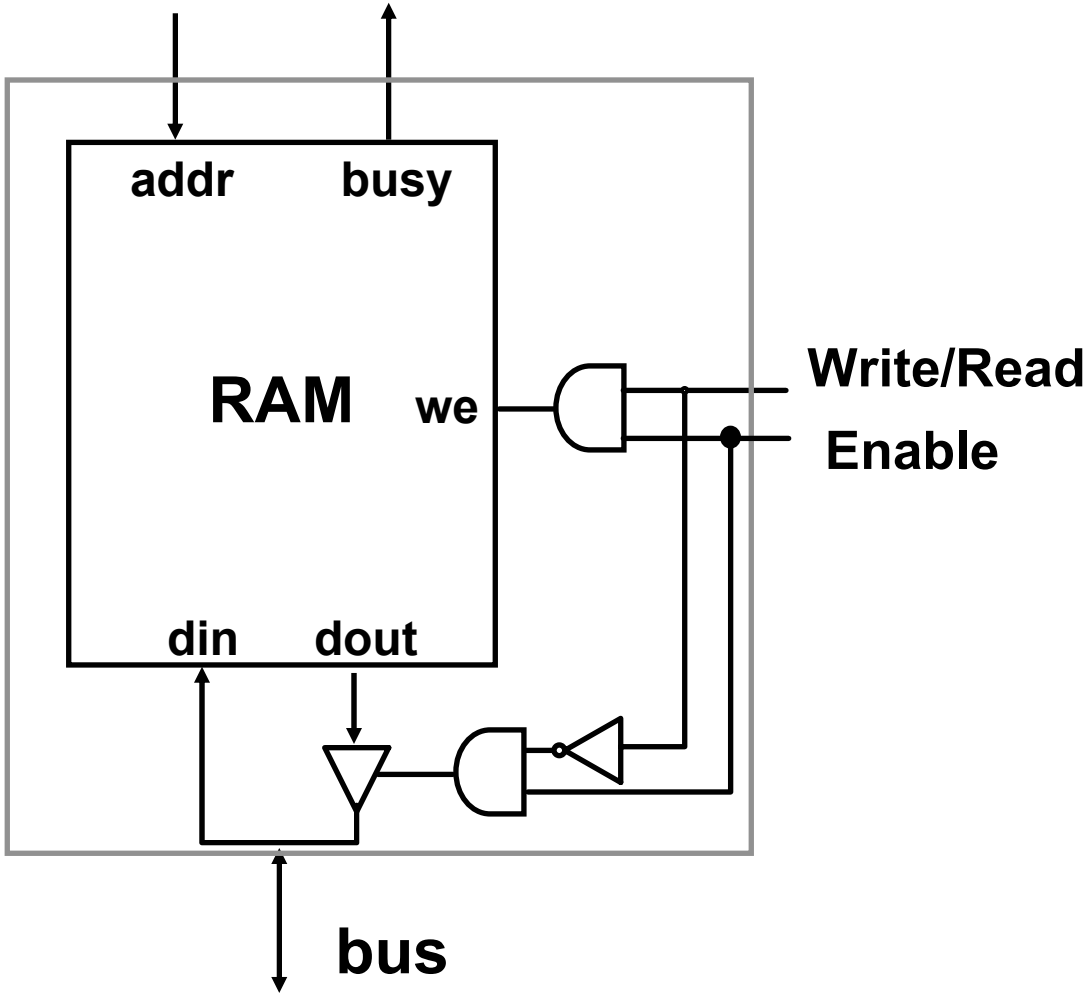



*Microinstruction: register to register transfer (17 control signals)*

MA ← PC *means* RegSel = PC; enReg=yes; IdMA= yes

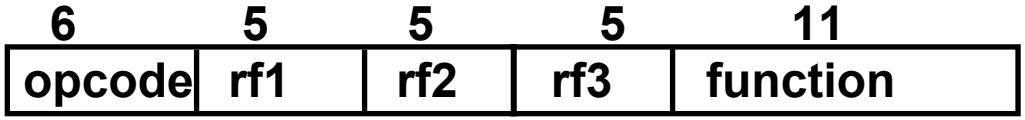
B ← Reg[rf2] *means* RegSel = rf2; enReg=yes; IdB = yes

# Memory Module

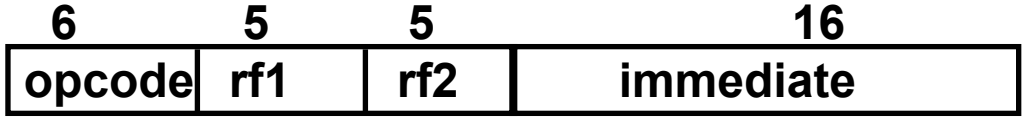



***We will assume that Memory operates asynchronously and is slow as compared to Reg-to-Reg transfers***

# DLX ALU Instructions




**Register-Register form:**
**Reg[rf3] ← function(Reg[rf1], Reg[rf2])**




**Register-Immediate form:**
**Reg[rf2] ← function(Reg[rf1], SignExt(immediate))**



# Instruction Execution

Execution of a DLX instruction involves

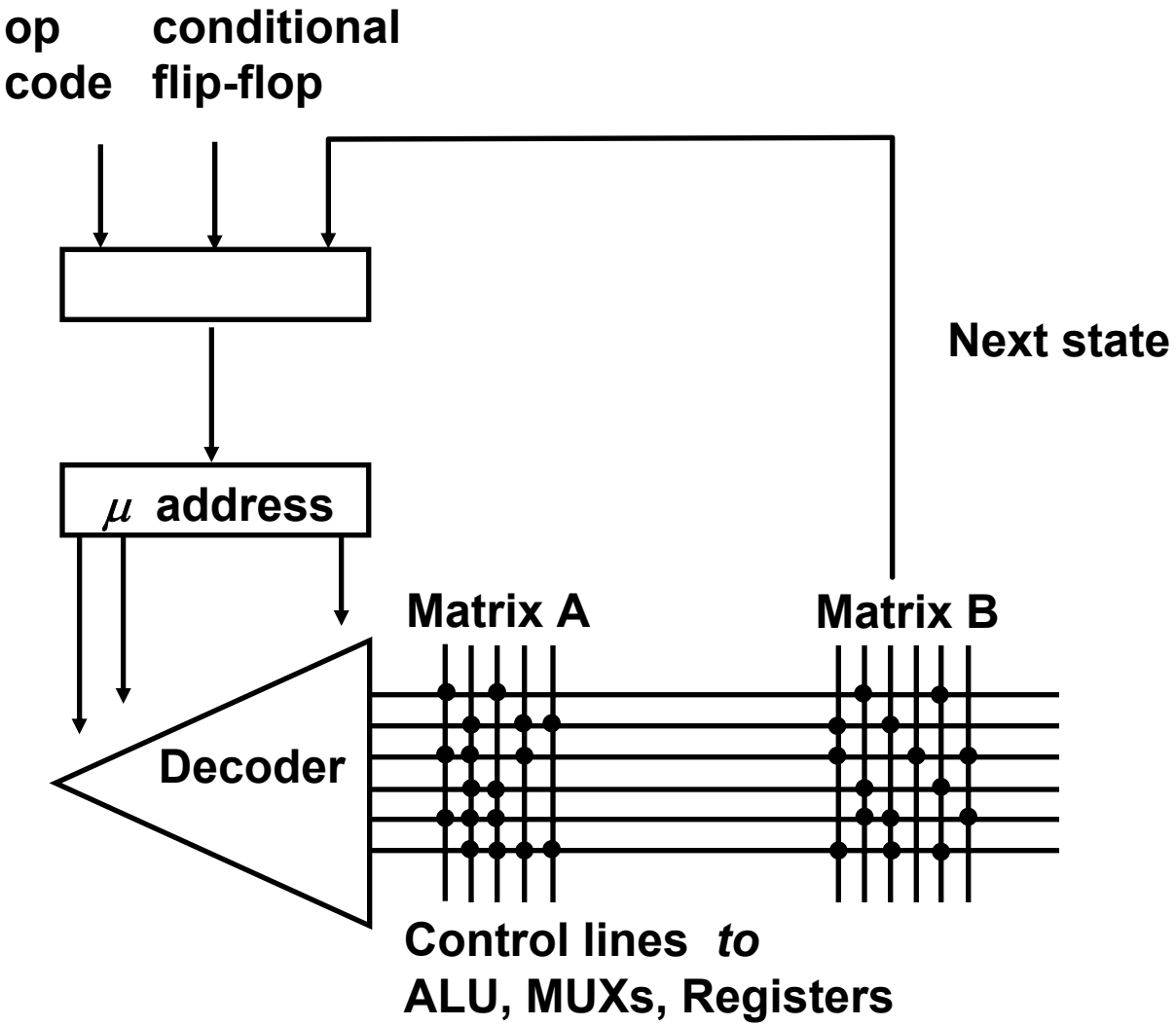
1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back to register file (optional)

and the computation of the address of the *next instruction*

# Microcontrol Unit

## *Maurice Wilkes, 1954*

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



# Microprogram Fragments

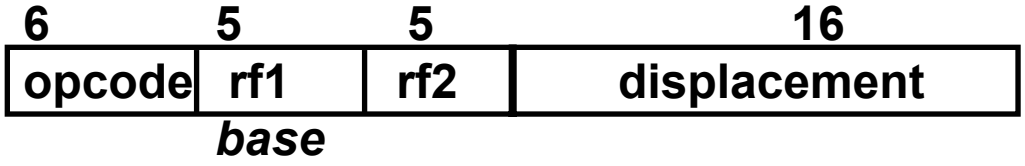
**instr fetch:**
MA ← PC
IR ← Memory
A ← PC
PC ← A + 4
dispatch on OPcode

*can be treated as a macro*

**ALU:**
A ← Reg[rf1]
B ← Reg[rf2]
Reg[rf3] ← func(A,B)
*do* instruction fetch

**ALUi:**
A ← Reg[rf1]
B ← Imm *sign extension ...*
Reg[rf2] ← Opcode(A,B)
*do* instruction fetch

# DLX Load/Store Instructions




## *Load/store byte, halfword, word to/from GPR:*

**LB, LBU, SB, LH, LHU, SH, LW, SW**

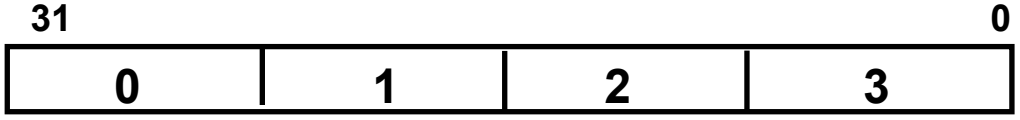
**byte and half-word can be sign or zero extended**

## *Load/store single and double FP to/from FPR:*

**LF, LD, SF, SD**

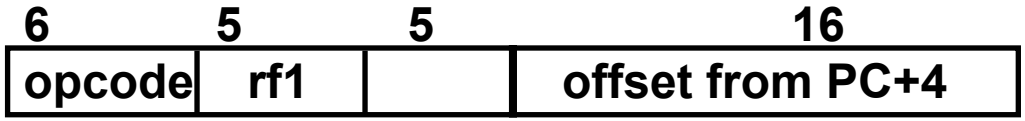
- **Byte addressable machine**
- **Memory access must be data aligned**
- **A single addressing mode**

  **(base) + displacement**
- **Big-endian byte ordering**



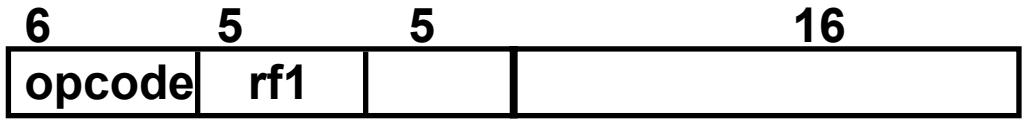
# DLX Control Instructions

## *Conditional branch on GPR*




BEQZ, BNEZ

## *Unconditional register-indirect jumps*




JR, JALR

## *Unconditional PC-relative jumps*


J, JAL

- PC-offset are specified in bytes
- jump-&-link (JAL) stores PC+4 into the link register (R31)
- *(Real DLX has delayed branches – ignored this lecture)*

# Microprogram Fragments *(cont.)*



**LW:** A ← Reg[rf1]
B ← Imm
MA ← A + B
Reg[rf2] ← Memory
*do* instruction fetch

**J:** A ← PC
B ← Imm
PC ← A + B
*do* instruction fetch

**beqz:** A ← Reg[rf1]
*If* zero?(A) *then go to* bz-taken
*do* instruction fetch

**bz-taken:** A ← PC
B ← Imm
PC ← A + B
*do* instruction fetch

# DLX Microcontroller: *first attempt*