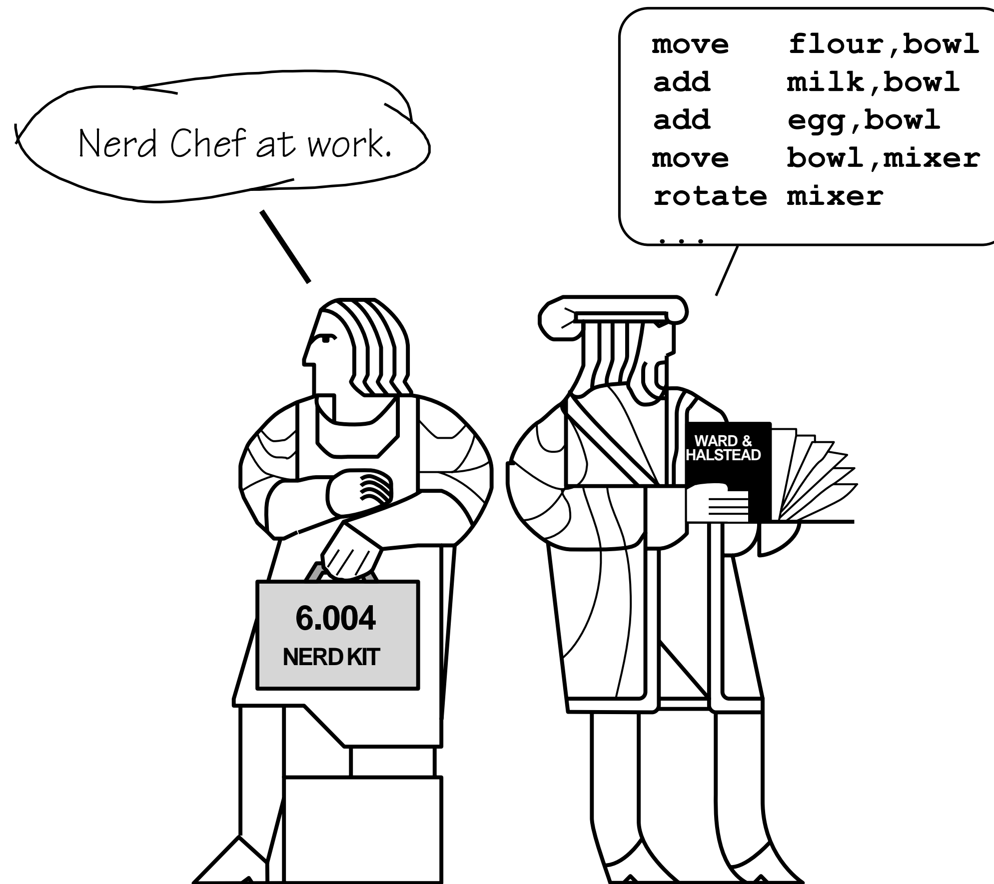
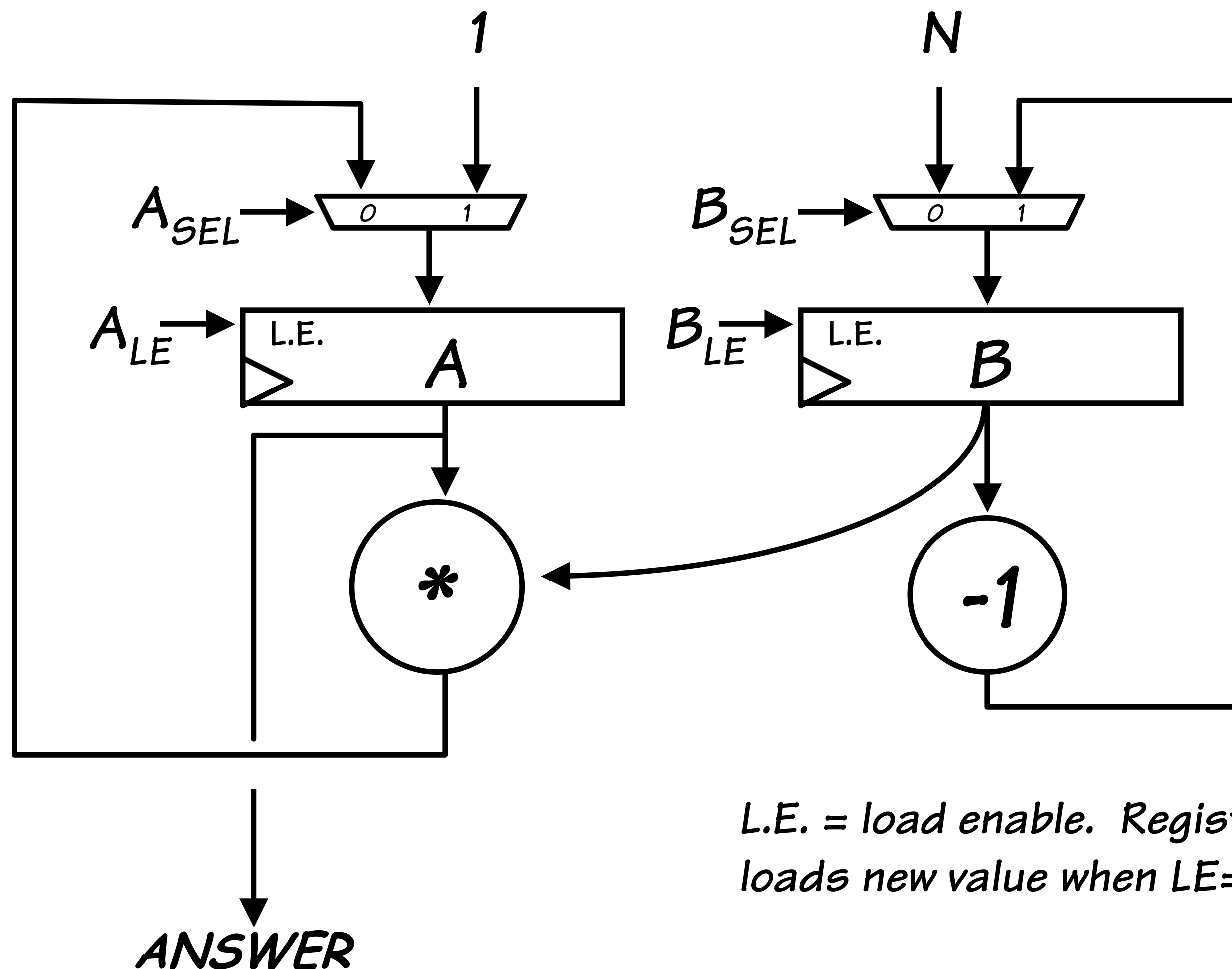
Designing an Instruction Set



Handouts: Lecture Slides, β docs

Let's build a simple computer

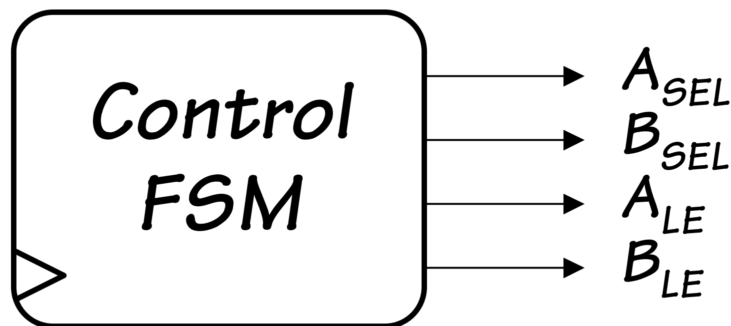
Data path for computing N*(N-1)

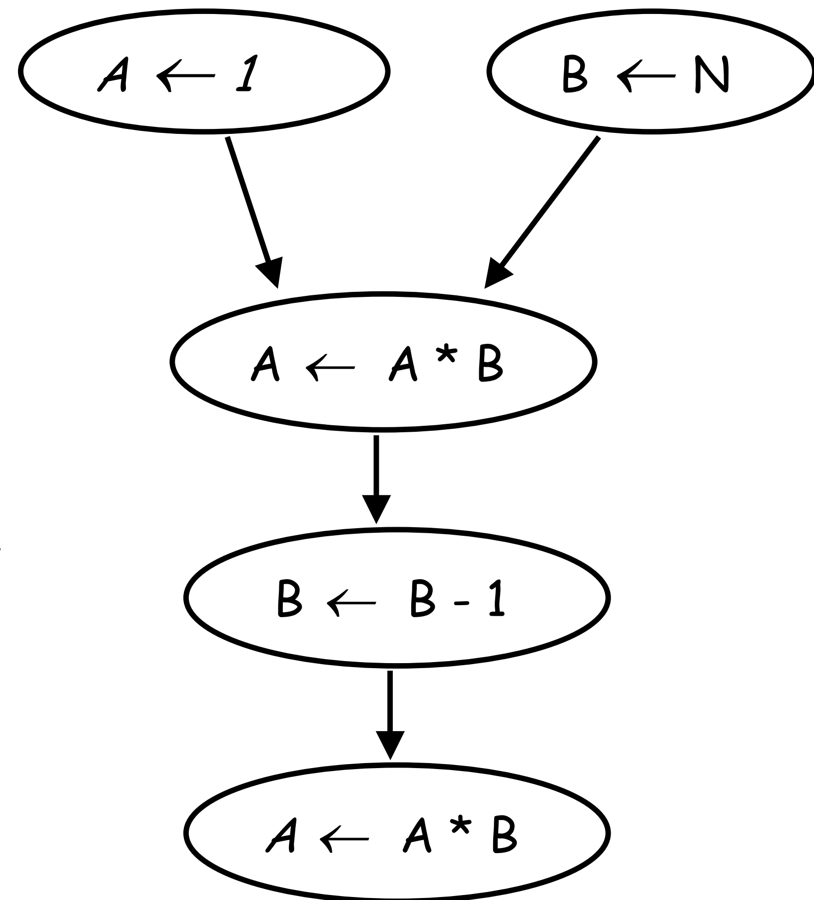


L.E. = load enable. Register only loads new value when LE=1

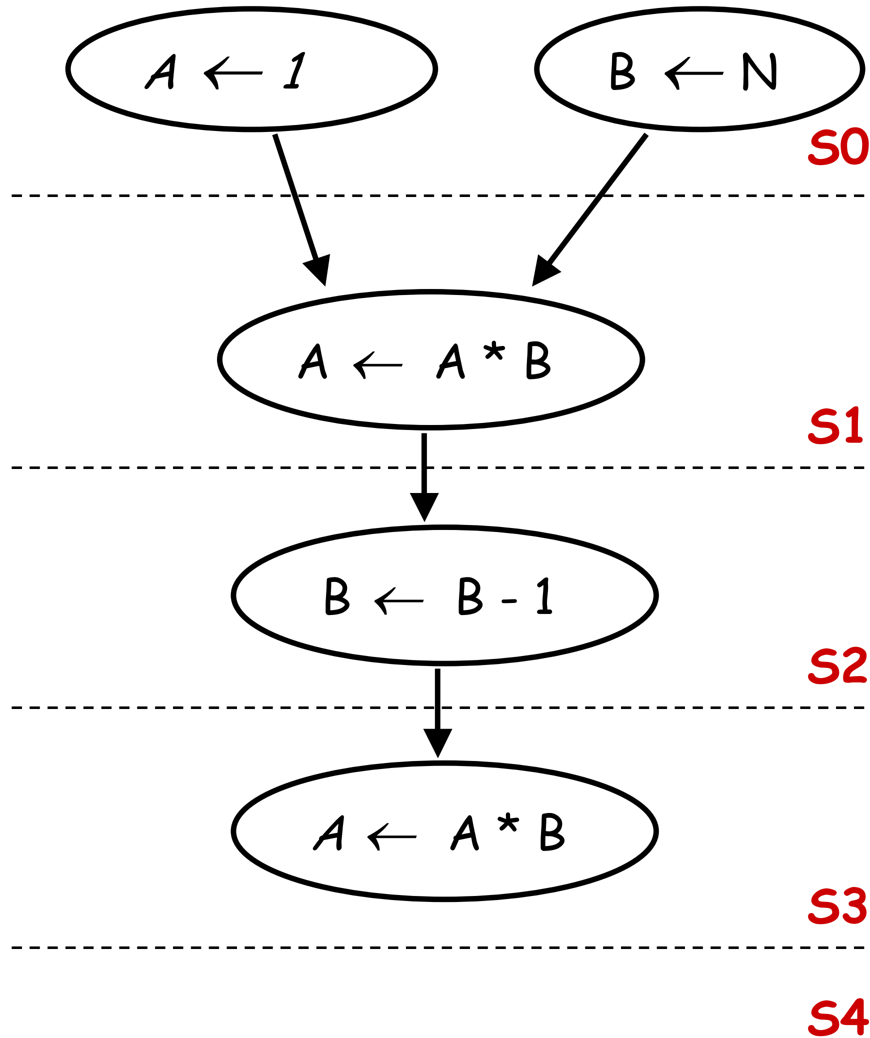
A programmable control system

Computing N*(N-1) with this data path is a multi-step process. We can control the processing at each step with a FSM. If we allow different control sequences to be loaded into the control FSM, then we allow the machine to be programmed.





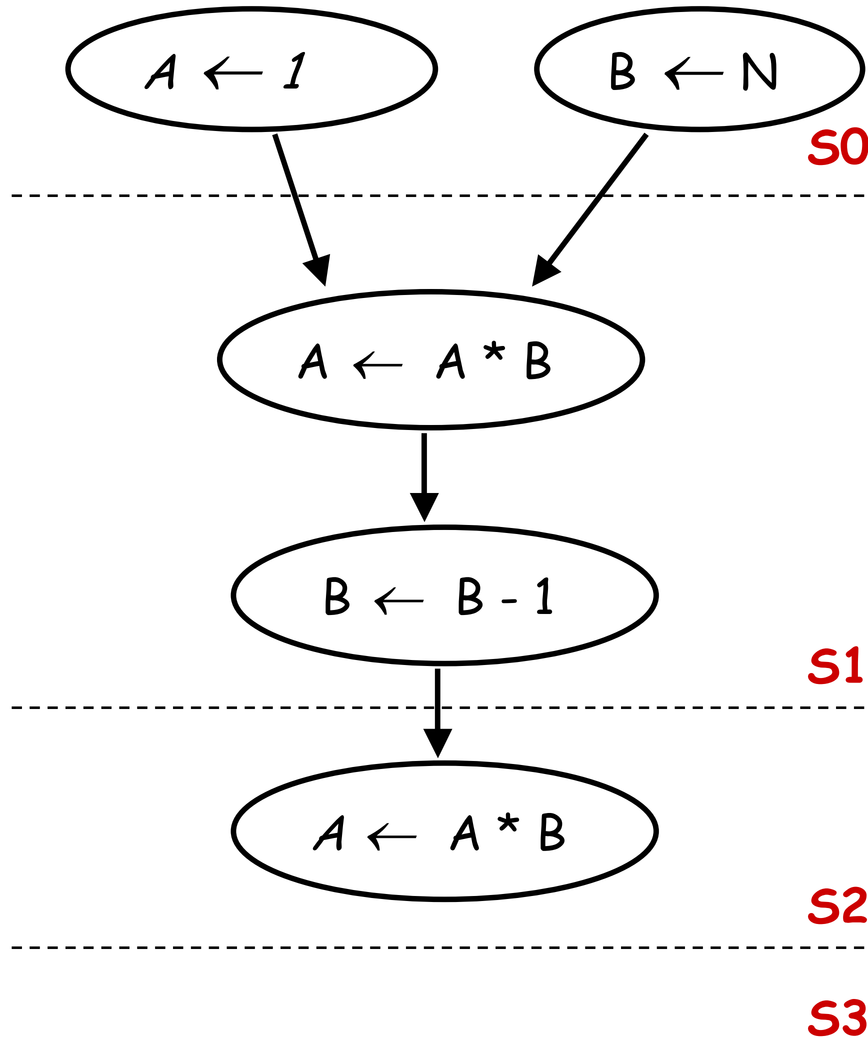
A first program



Once more, writing a control program is nothing more than filling in a table

S_N	S_{N+1}	A_{sel}	B_{sel}	A_{le}	B_{le}
0	1				
1	2				
2	3				
3	4				
4	4				

An optimized program



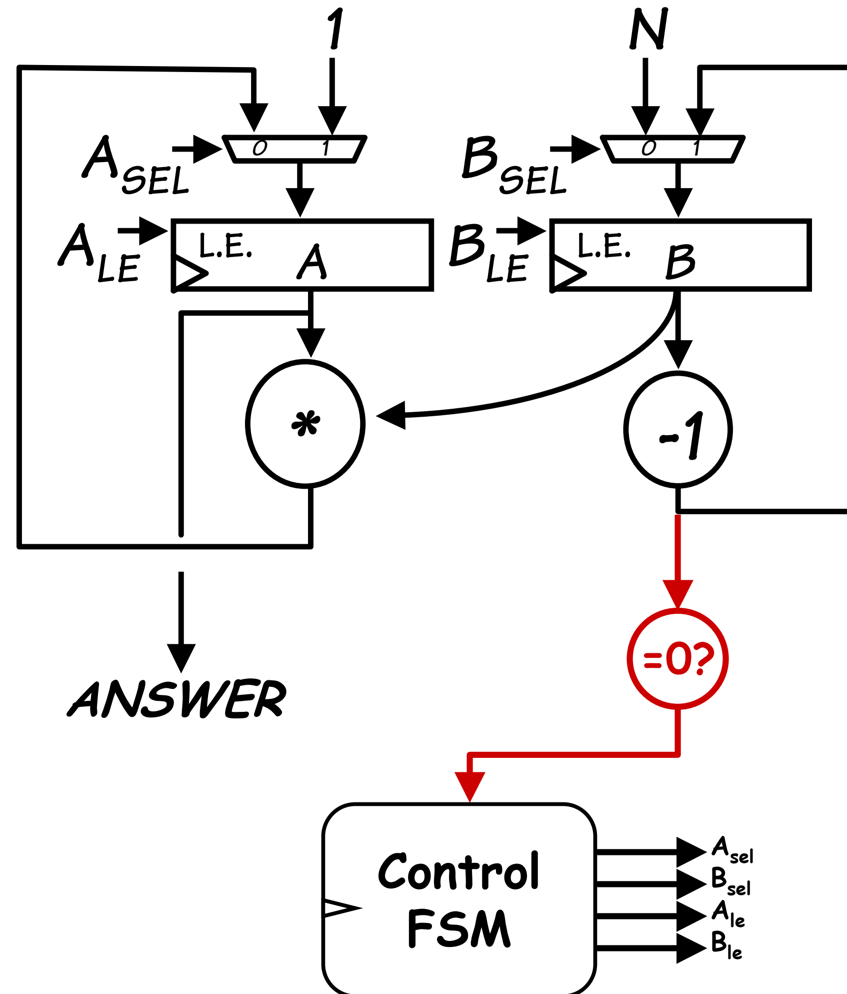
Some parts of the program can be computed simultaneously.

S_N	S_{N+1}	A_{sel}	B_{sel}	A_{le}	B_{le}
0	1				
1	2				
2	3				
3	3				

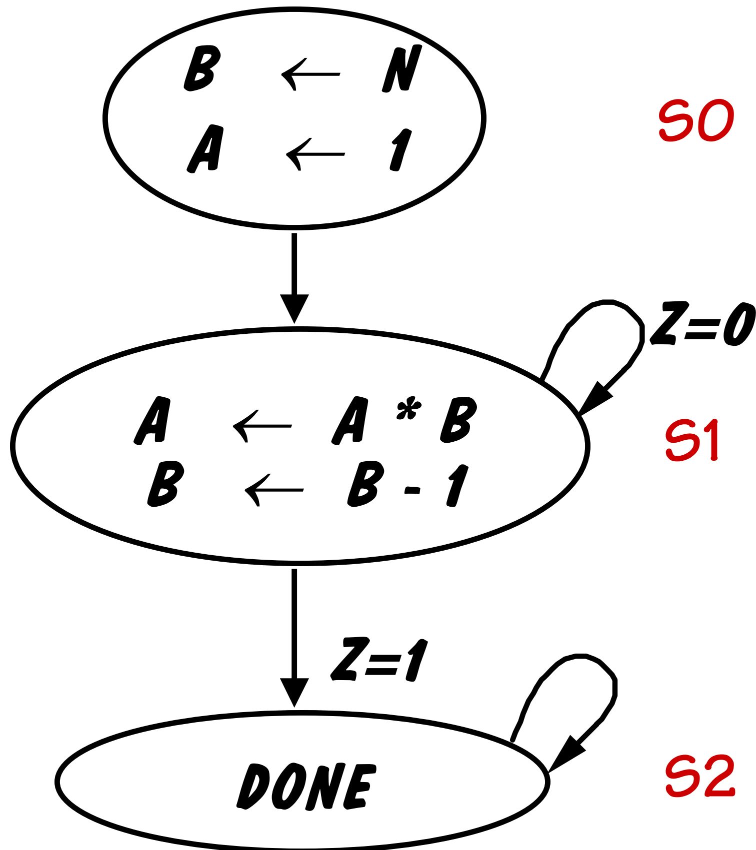
Computing Factorial

The advantage of a programmable control system is that we can reconfigure it to compute new functions.

In order to compute N! we will need to add some new logic and an input to our control FSM.



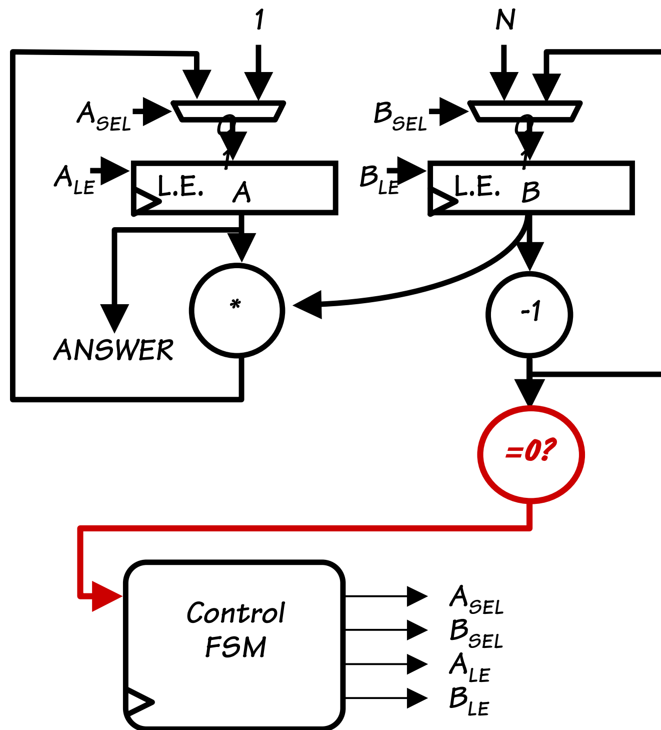
Program for Factorial



Programmability allows us to reuse data paths to solve new problems. What we need is a general purpose data path, which can be used to efficiently solve most problems as well as a easier way to control it.

Z	S_N	S_{N+1}	A_{sel}	B_{sel}	A_{le}	B_{le}
-	0					
0	1					
1	1					
-	2					

Factorial Engine




```
Z S | S' Asel Bsel Ale Ble
----+---------------------
- 0 | 1   1    0    1   1    A=1, B=N
0 1 | 1   0    1    1   1    A=A*B, B=B-1
1 1 | 2   0    1    1   1
- 2 | 2   -    -    0   0    done
```

The same data paths could compute N*(N-1), Factorial, ... only difference: *information in control ROM.*

Today's big idea:

general purpose computer architecture

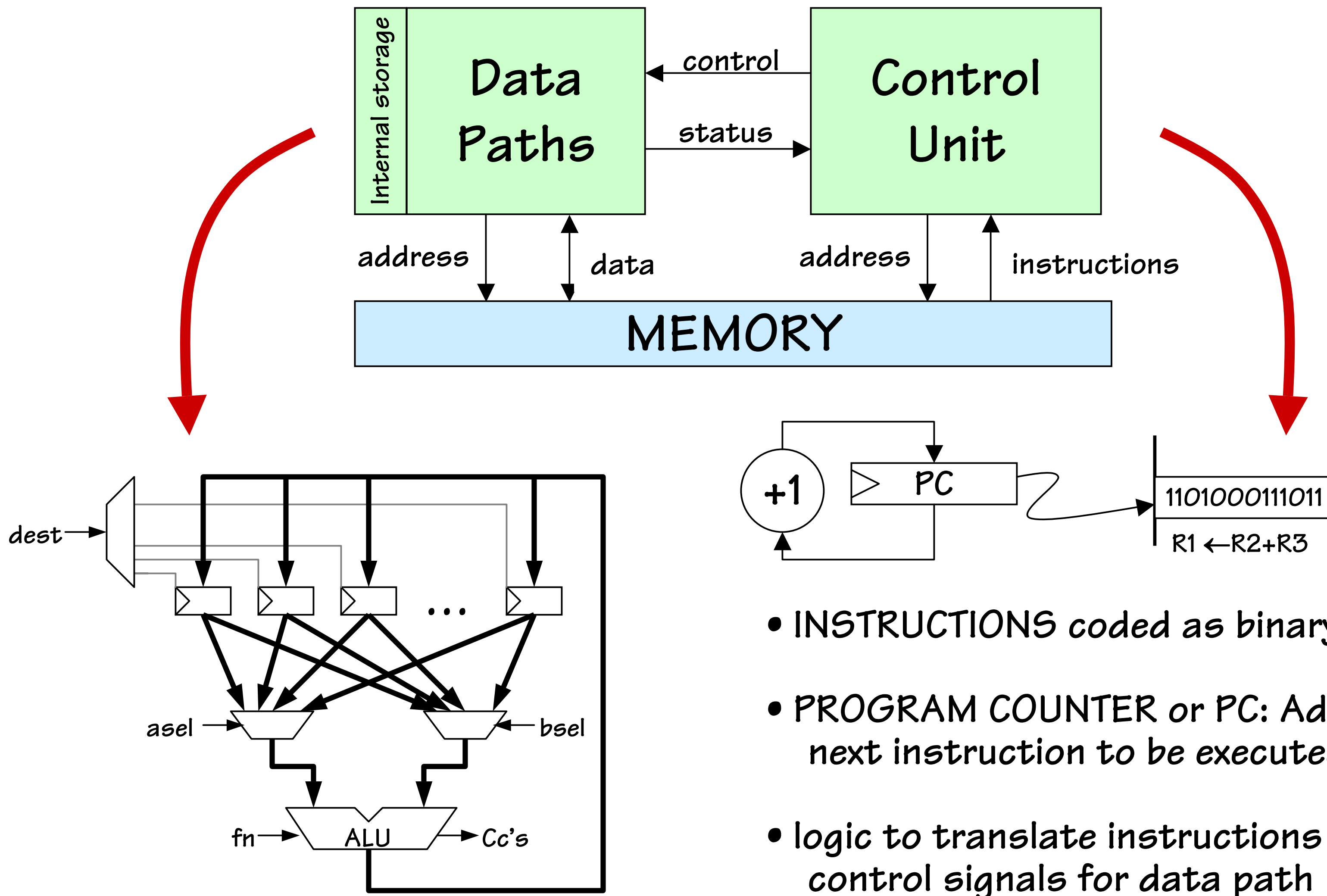
- One set of "UNIVERSAL" Data paths
- ENCODED sequence of operational steps dictate specific function to be performed...

the PROGRAM!

New Issue:

HOW to encode the Program?

Anatomy of an Interpreter



- INSTRUCTIONS coded as binary data
- PROGRAM COUNTER or PC: Address of next instruction to be executed
- logic to translate instructions into control signals for data path

Questions to be answered:

Data path questions:

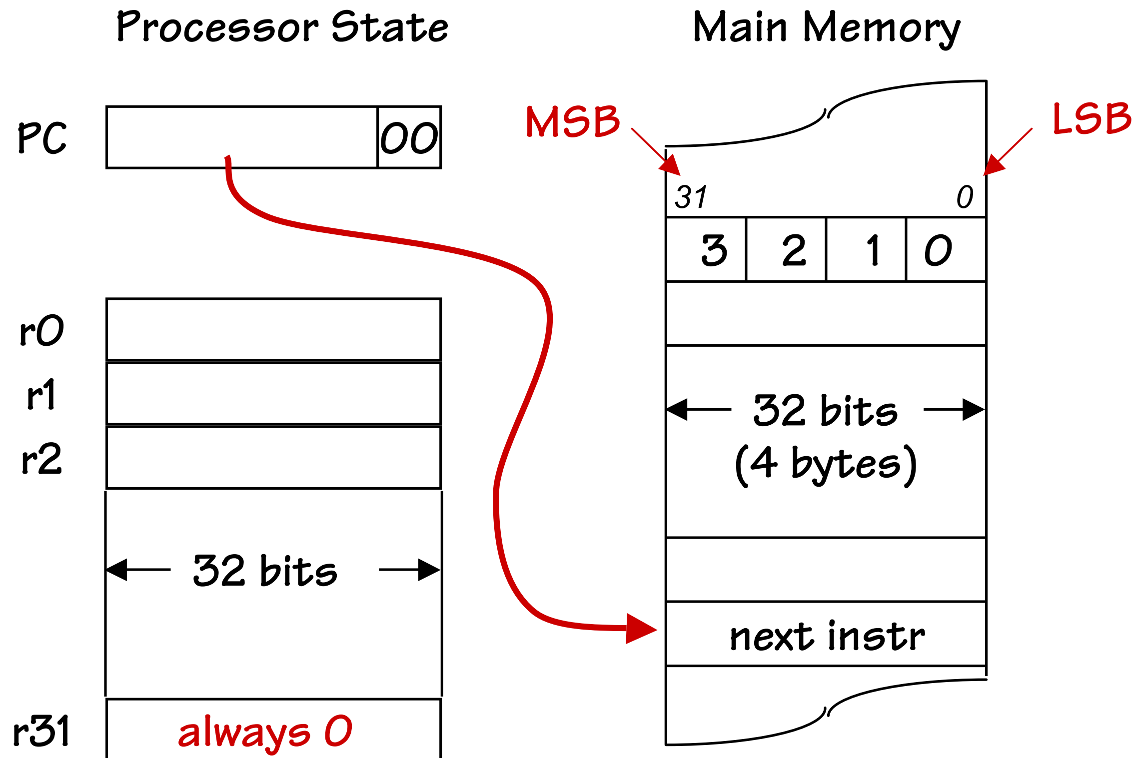
- how much internal storage?
- what are the ALU functions?
- provision for constant operands?
- how does data get to/from memory?
- width (in bits) of the registers/ALU?

Control unit questions:

- how should instructions be encoded?
  - low-level (eg, ctl signals for data path)
  - high-level (eg, "fill polygon")
  - Huffman encoded (so commonly-used insts are short)
  - etc., etc., etc.

next	fn	dest	asel	bsel

β Model of Computation



Fetch/Execute loop:

- fetch Mem[PC]
- PC = PC + 4†
- execute fetched instruction (may change PC!)
- repeat!

MSB = most significant bit
LSB = least significant bit

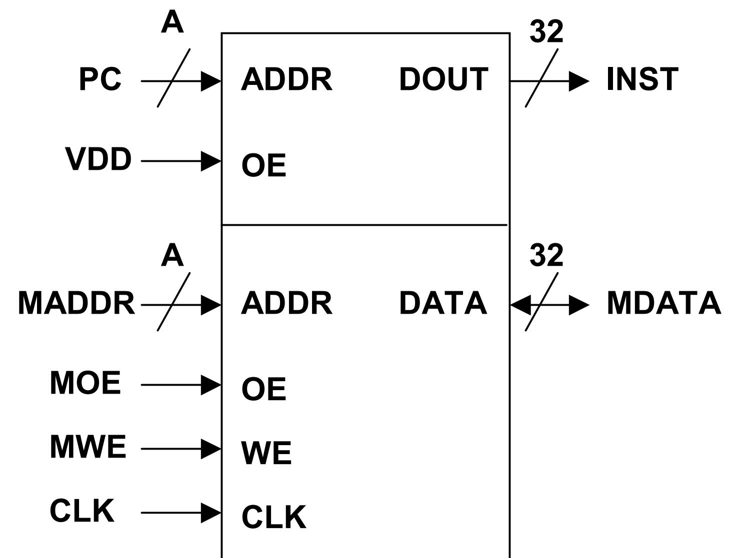
†Even though each memory word is 32-bits wide, for historical reasons the β uses byte memory addresses. Since each word contains four 8-bit bytes, addresses of consecutive words differ by 4.

β "Main" Memory

Instructions and data are initially stored in *main memory* (so called in order to distinguish it from other, smaller, special-purpose memories we'll learn about later).

- memories consist of many *locations* (aka words), each of which has some number of bits. In the β each memory location has 32 bits.
- memories have one or more access *ports* (multiport memories can access more than one location at a time)
- each port includes
  - an *address* that specifies a location
  - a *data bus* that supplies write data or receives read data (or both!)
  - *control signals* indicating when the memory should perform an access

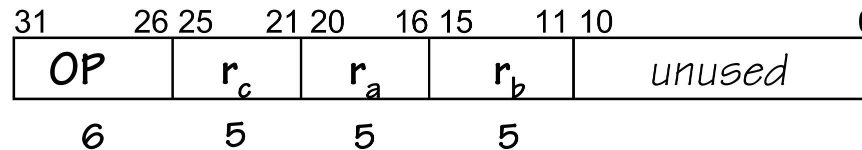
Dual-port
32-bit
Main memory



ADDR = selects one of 2^A locations
OE = output enable for READ data
WE = write enable for WRITE data
CLK = data written on rising edge

β ALU Operations

What the machine sees: 32-bit instruction word



6-bit operation field can specify one of 64 operations:

- arithmetic: ADD, SUB, MUL, DIV
- compare: CMPEQ, CMPLT, CMPLE
- boolean: AND, OR, XOR
- shift: SHL, SHR, SAR

5-bit register fields can name one of 32 registers:

- Ra and Rb are the operands,
- Rc is the destination.
- R31 reads as 0, unchanged by writes

What we prefer to see: symbolic ASSEMBLY LANGUAGE

ADD(ra, rb, rc)

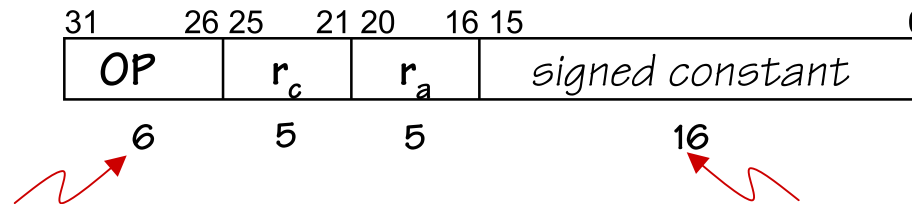
If rc is 31, the result is discarded and the instruction does nothing ("NOP")

Reg[rc] = Reg[ra] + Reg[rb]

"Add the contents of ra to the contents of rb; store the result in rc"

β ALU Operations w/ constant

Alternative instruction format w/ built-in constant:



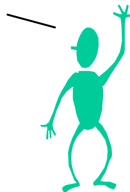
arithmetic: ADDC, SUBC, MULC, DIVC
compare: CMPEQC, CMPLTC, CMPLEC
boolean: ANDC, ORC, XORC
shift: SHLC, SHRC, SARC

Two's complement 16-bit constant for numbers from –32768 to 32767; sign-extended to 32 bits before use.

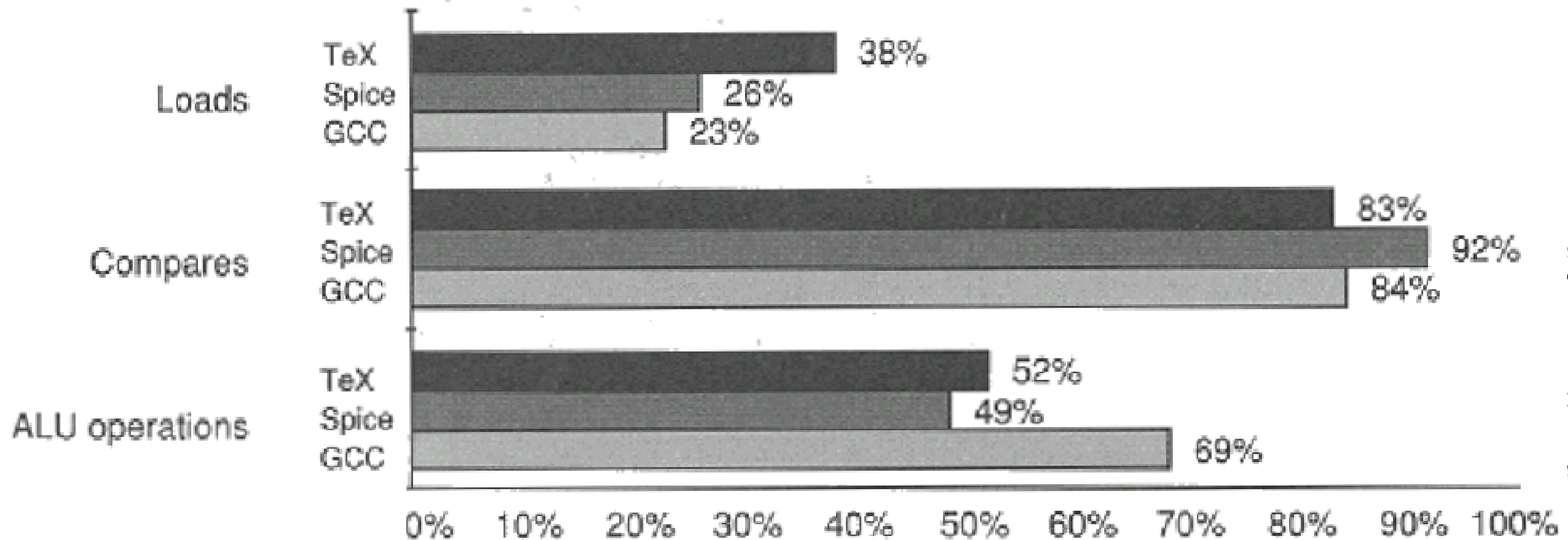
ADDC(ra, const, rc)   Reg[rc] = Reg[ra] + sxt(const)

"Add the contents of ra to const; store the result in rc"

Wait! Do we really need this extra complication?



Do we need built-in constants?



Percentage of the operations that use a constant operand

One way to answer architectural questions is to evaluate the consequences of different choices using carefully chosen representative benchmarks (programs and/or code sequences). Make choices that are "best" according to some metric (cost, performance, ...).

β Loads & Stores

LD(ra, const, rc)   Reg[rc] = Mem[Reg[ra] + sxt(const)]

"Fetch into rc the contents of the memory location whose address is C plus the contents of ra"

Abbreviation: LD(C, rc) for LD(R31, C, rc)

ST(rc, const, ra)   Mem[Reg[ra] + sxt(const)] = Reg[rc]

"Store the contents of rc into the memory location whose address is C plus the contents of ra"

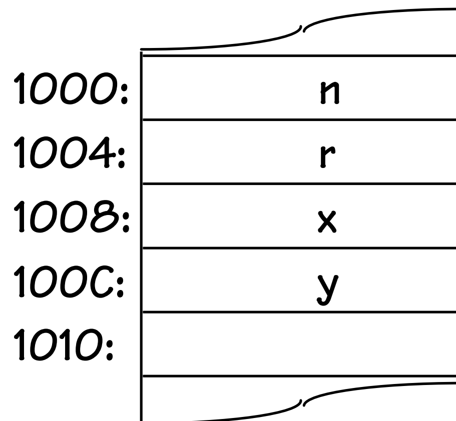
Abbreviation: ST(rc,C) for ST(rc, C, R31)

BYTE ADDRESSES, but only 32-bit word accesses to word-aligned addresses are supported. Low two address bits are ignored!

Storage Conventions

Addr assigned at compile time

- Variables live in memory
- Operations done on registers
- Registers hold Temporary values




```
int x, y;
y = x * 37;
```

Compilation approach:
LOAD, COMPUTE, STORE

translates to

```
LD(r31, 0x1008, r0)
MULC(r0, 37, r0)
ST(r0, 0x100C, r31)
```

or, more humanely, to

```
x=0x1008
y=0x100C
LD(x, r0)
MULC(r0, 37, r0)
ST(r0, y)
```

Ra defaults to R31 (0)

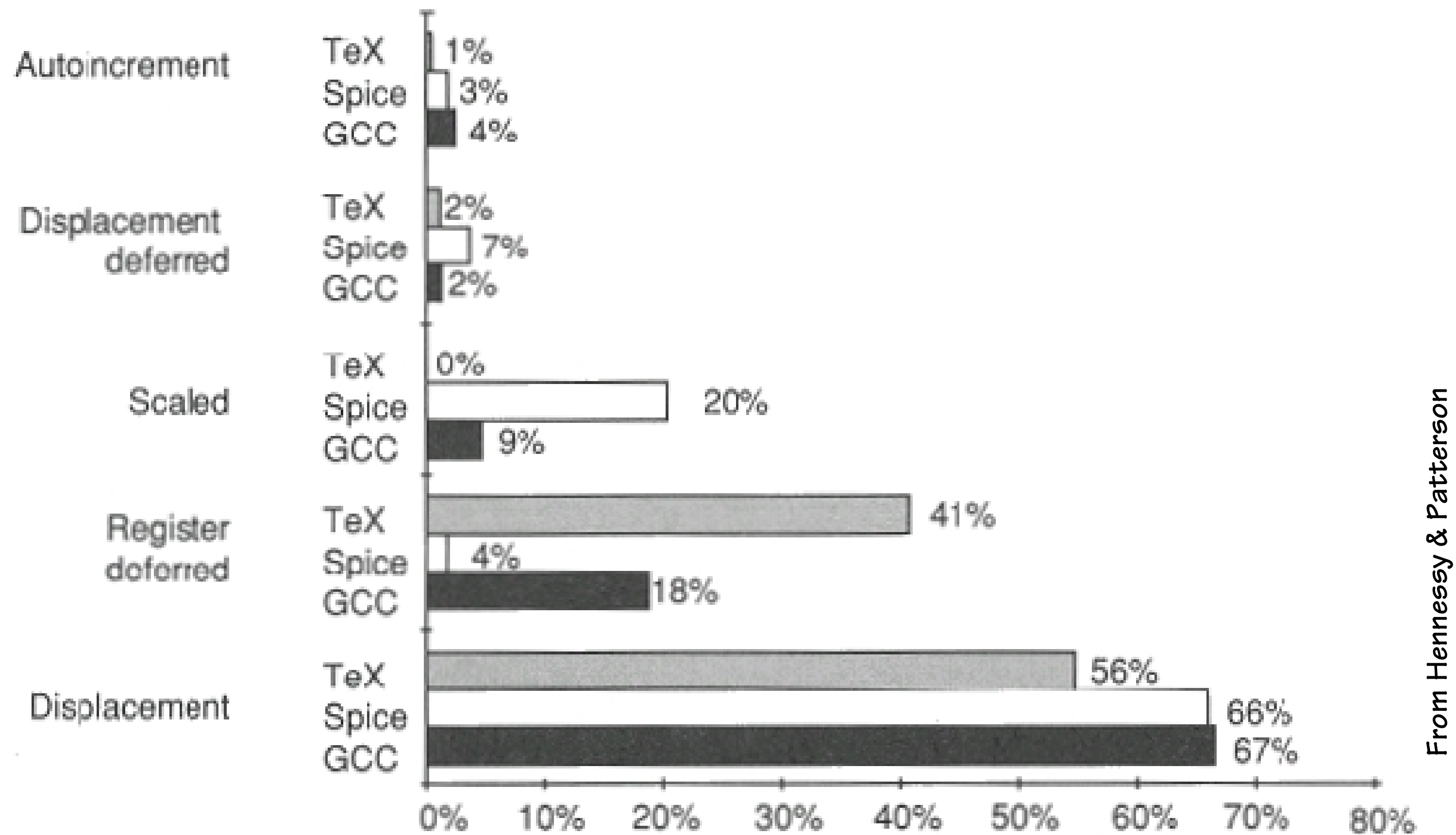
Alternative Addressing Modes

β can do these with appropriate choices for Ra and const

- **Absolute**: "(constant)"
  - Value = Mem[constant]
  - Use: accessing static data
- **Indirect** (aka Register deferred): "(Rx)"
  - Value = Mem[Reg[x]]
  - Use: pointer accesses
- **Displacement**: "constant(Rx)"
  - Value = Mem[Reg[x] + constant]
  - Use: access to local variables
- **Indexed**: "(Rx + Ry)"
  - Value = Mem[Reg[x] + Reg[y]]
  - Use: array accesses (base+index)
- **Memory indirect**: "@(Rx)"
  - Value = Mem[Mem[Reg[x]]]
  - Use: access thru pointer in mem
- **Autoincrement**: "(Rx)+"
  - Value = Mem[Reg[x]]; Reg[x]++
  - Use: sequential pointer accesses
- **Autodecrement**: "-(Rx)"
  - Value = Reg[X]--; Mem[Reg[x]]
  - Use: stack operations
- **Scaled**: "constant(Rx)[Ry]"
  - Value = Mem[Reg[x] + c + d*Reg[y]]
  - Use: array accesses (base+index)

Argh! Need a cost/benefit analysis!

Memory Operands: usage



From Hennessy & Patterson

Usage of different memory operand modes

Translation of an Expression

```
int x, y;
y = (x-3)*(y+123456)

x:      long(0)
y:      long(0)
c:      long(123456)
    ...
    LD(x, r1)
    SUBC(r1,3,r1)
    LD(y, r2)
    LD(C, r3)
    ADD(r2,r3,r2)
    MUL(r2,r1,r1)
    ST(r1,y)
```

- VARIABLES are allocated storage in main memory
- VARIABLE references translate to LD or ST
- OPERATORS translate to ALU instructions
- SMALL CONSTANTS translate to ALU instructions w/ built-in constant
- "LARGE" CONSTANTS translate to initialized variables

Summary

- Instructions and data are stored in 32-bit wide main memory
- PC contains address of next instruction to be executed
  - normally incremented (by 4) after each instruction fetch
  - BR/JMP instructions save PC and then modify it
  - BR's use PC-relative addressing with a word displacement
- Instructions are decoded by control unit
  - 6-bit opcode, 5-bit register fields
  - 3-reg format: Reg[rc] = Reg[ra] *op* Reg[rb]
  - 2-reg + constant format: Reg[rc] = Reg[ra] *op* sxt(const)
    - LD/ST: address is Reg[ra] + sxt(const)
    - BR: const is used as pc-relative word displacement
- Compilation strategy
  - load operands from memory into regs
  - compute using reg operands (+ small consts), reg destination
  - store result into memory
- ISA design requires tradeoffs, usually based on benchmark results