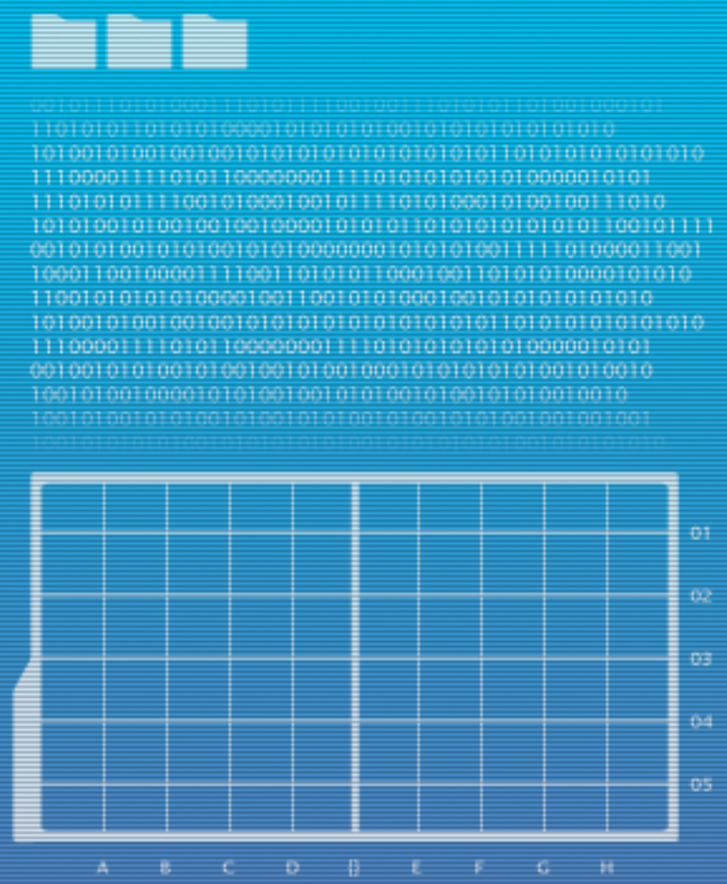
DEPARTMENT OF
INFORMATION SYSTEMS
AND COMPUTER SCIENCE

Machine Language and Assembly Programming

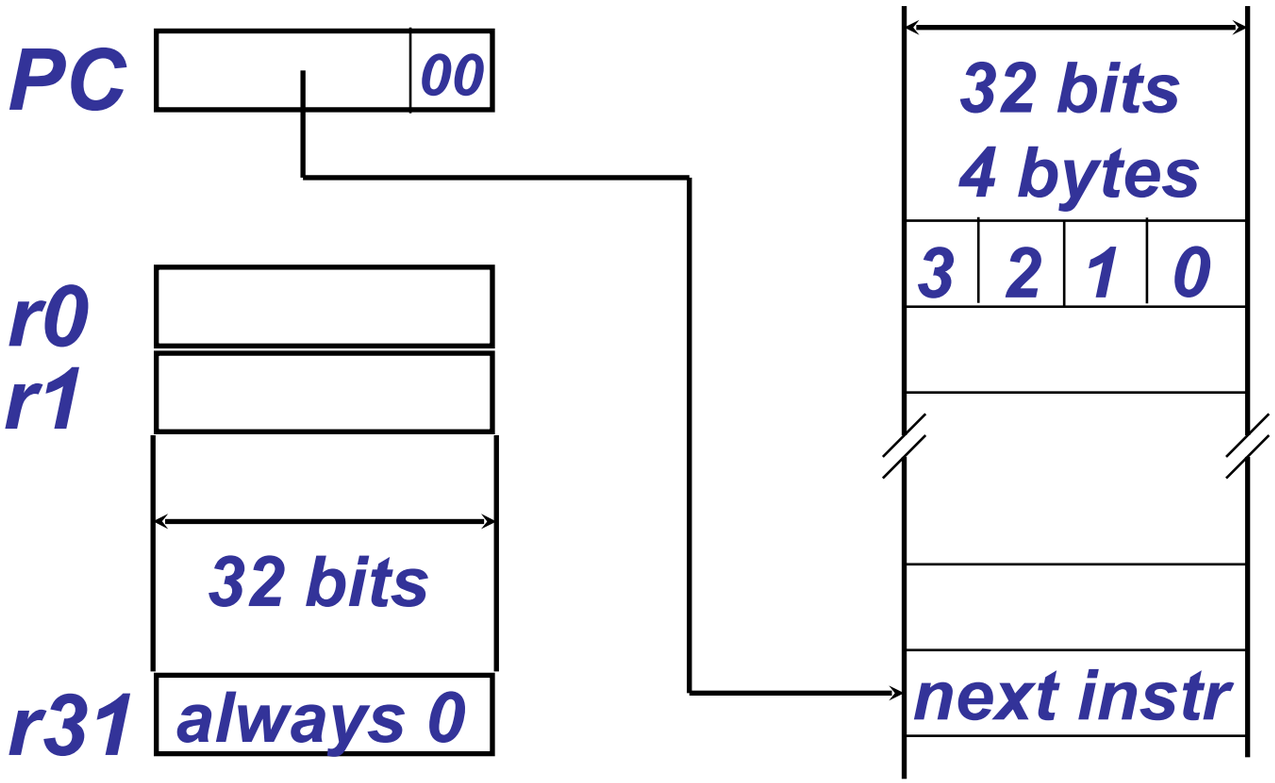
```
z = x + y;
// as simple as it looks?
```



So... Now What?

- We can now build a CPU!
- What do we need?
  - ALU
  - General Purpose Registers (R0-R31)
    - Temporary storage for operations.
    - Memory for storing code and data.
  - A Program Counter (PC) to remember where we are in the code.
  - An FSM to control everything.
    - The FSM “interprets” the “instructions”, which are stored in memory.

A CPU



General Purpose Registers

Fetch/Execute Loop

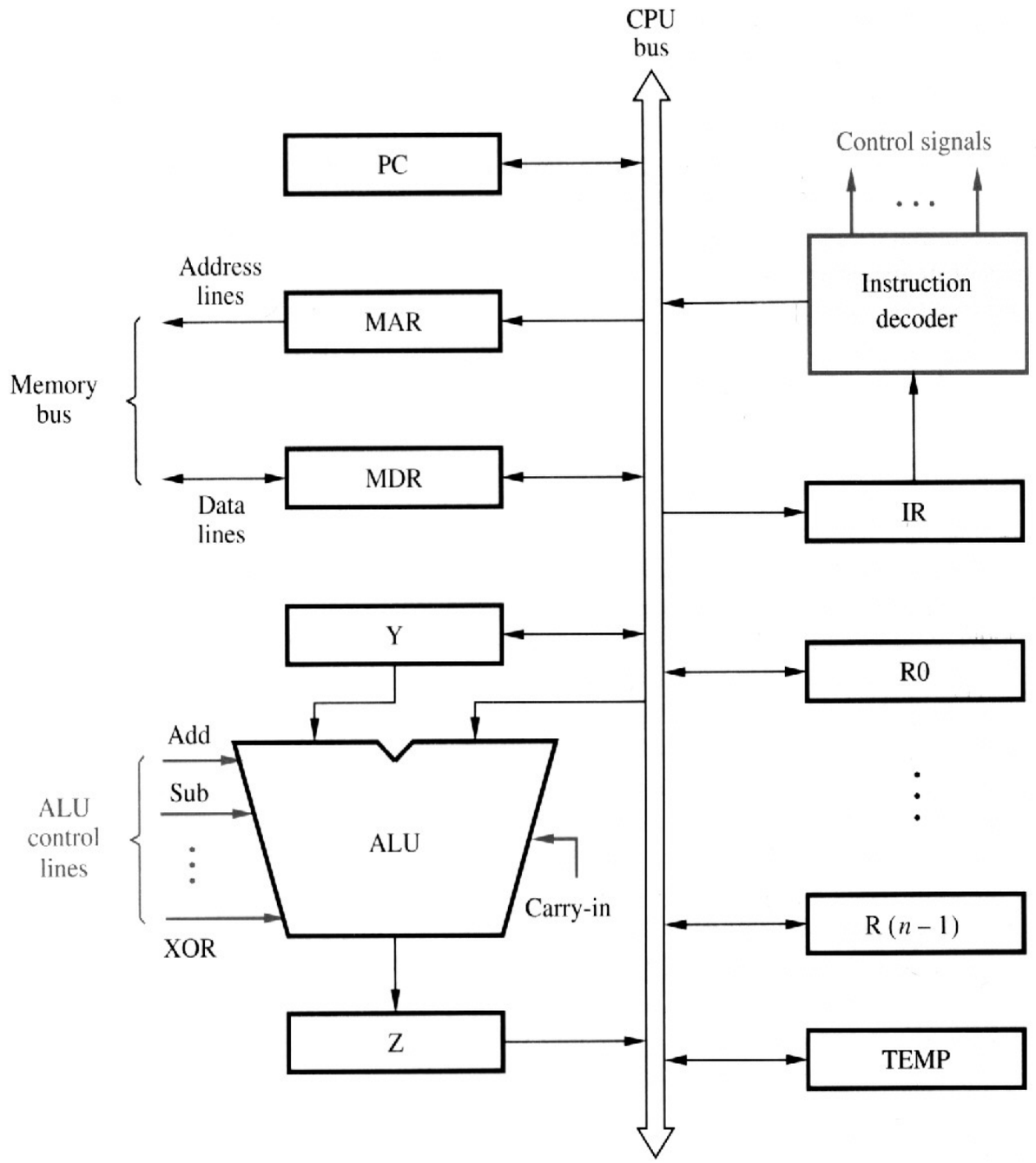
Fetch <PC>

PC ← <PC> + 4

Execute fetched instruction

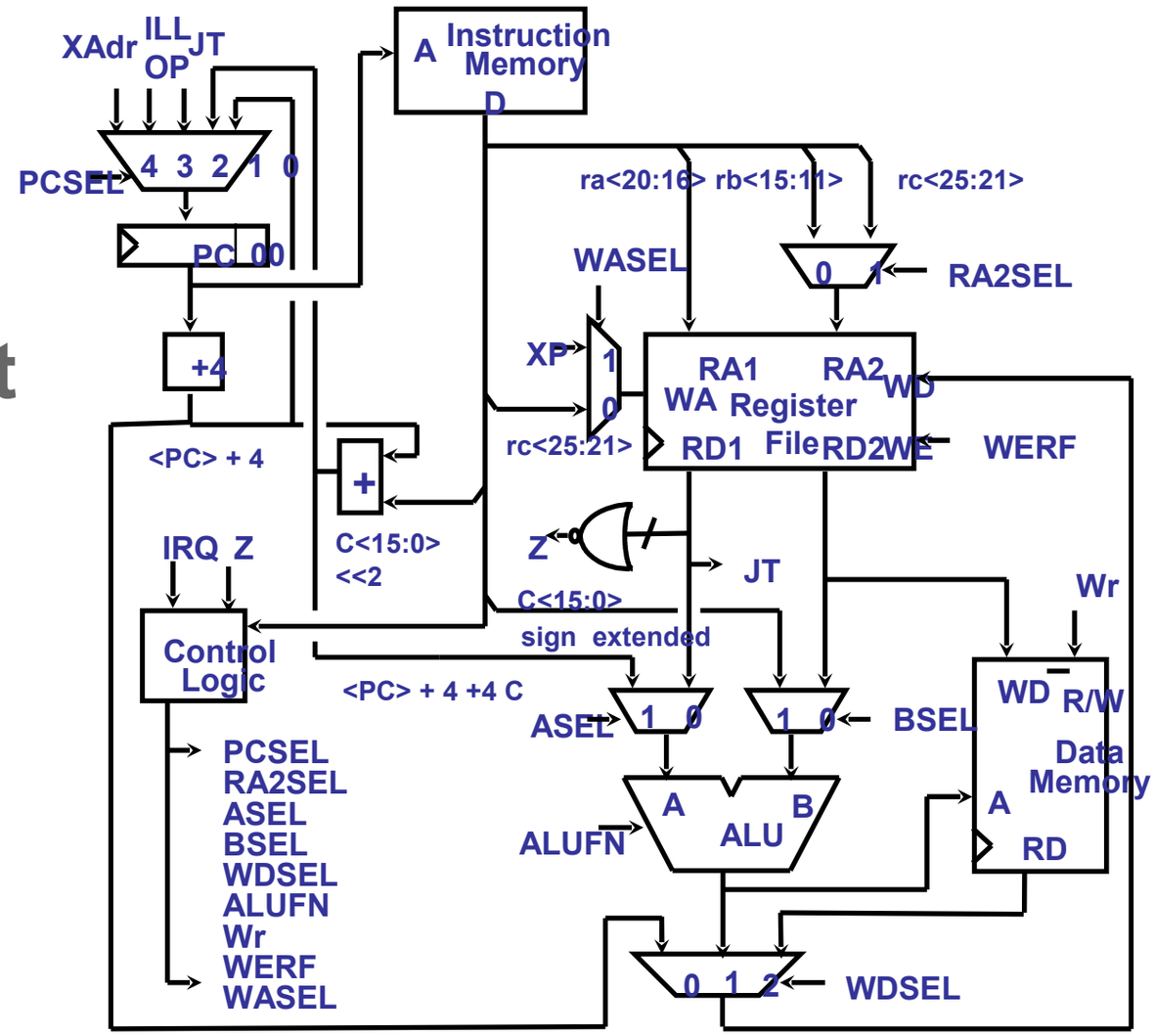
Bus-based CPU

- Components share a “bus” or a set of wires.
  - “Tri-state drivers” control who “drive” or control bus inputs and outputs at any given time.
- Saves wire and space, BUT limits data movement.
- Takes several cycles to execute an instruction:
  - Fetch instruction.
  - Load operands.
  - Operate on data.
  - Store data.
- Microprogramming
  - Different instructions can take different # of cycles.
  - Allows complex instructions.



Non-Bus-Based CPU

- Components are connected directly.
- Much more wires and space.
  - 32 or 64 bits per set of wires.
- But more things can happen at the same time.
- Only 1 cycle per instruction.
- "Hardwired Control"
  - Each instruction corresponds to the setting of signals (e.g., PCSEL, BSEL, etc.) for 1 clock cycle.
  - Simpler, and faster, but cannot do complex instructions (in 1 cycle).



Microprogramming vs. Hardwired Control

- Microprogramming
  - Very flexible, can do complex instructions.
    - We can do instructions with conditionals and loops, like COPYARRAY, or even SQRT and FACTORIAL!
  - Takes several cycles per instruction.
    - Leads to CISC (Complex Instruction Set Computer).
- Hardwired control
  - Simpler to implement and faster.
  - Limited to one-cycle instructions (no loops).
  - Leads to RISC (Reduced Instruction Set Computer).

CISC vs. RISC

- CISC (Complex Instruction Set Computer)
  - More functionality per instruction.
  - Shorter code.
  - But variable time per instruction.
- RISC (Reduced Instruction Set Computer)
  - Simple functionality.
  - Longer code.
  - But 1-cycle execution = Can be pipelined!
- Hybrid (e.g., Pentium 4, Intel Core i7, etc.)
  - Support a CISC instruction set, but internally split it into RISC-like 1-cycle instructions (usually in addition to other optimizations such as using these instructions in parallel).
- In this class, we will do RISC (aka Load-Store Architecture)!

How do we program a CPU?

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Control Signal Specification

```
z=x+y;
```

```
LD(R31,X,r1)
LD(R31,Y,r2)
ADD(r1,r2,r3)
ST(r3,Z,R31)
```

```
0110 0000 0011 1111 0000 0010 0000 0000
0110 0000 0101 1111 0000 0010 0000 0100
1000 0000 0110 0001 0001 0000 0000 0000
0110 0100 0111 1111 0000 0010 0000 1000
```

```
ALUOP[0:3] <= InstReg[9:11] & MASK
```

Instruction Set Design

software

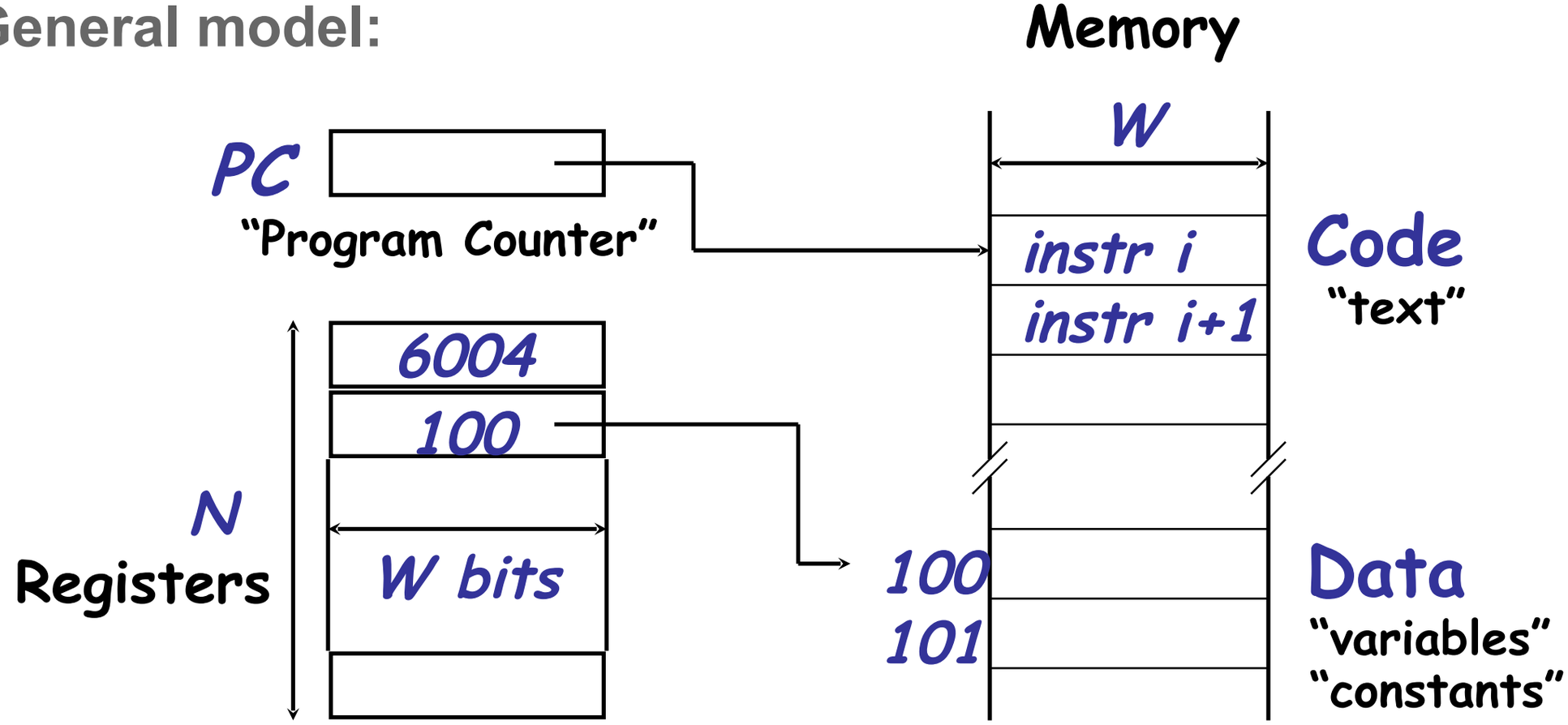
instruction set

hardware

Which is easier to change?

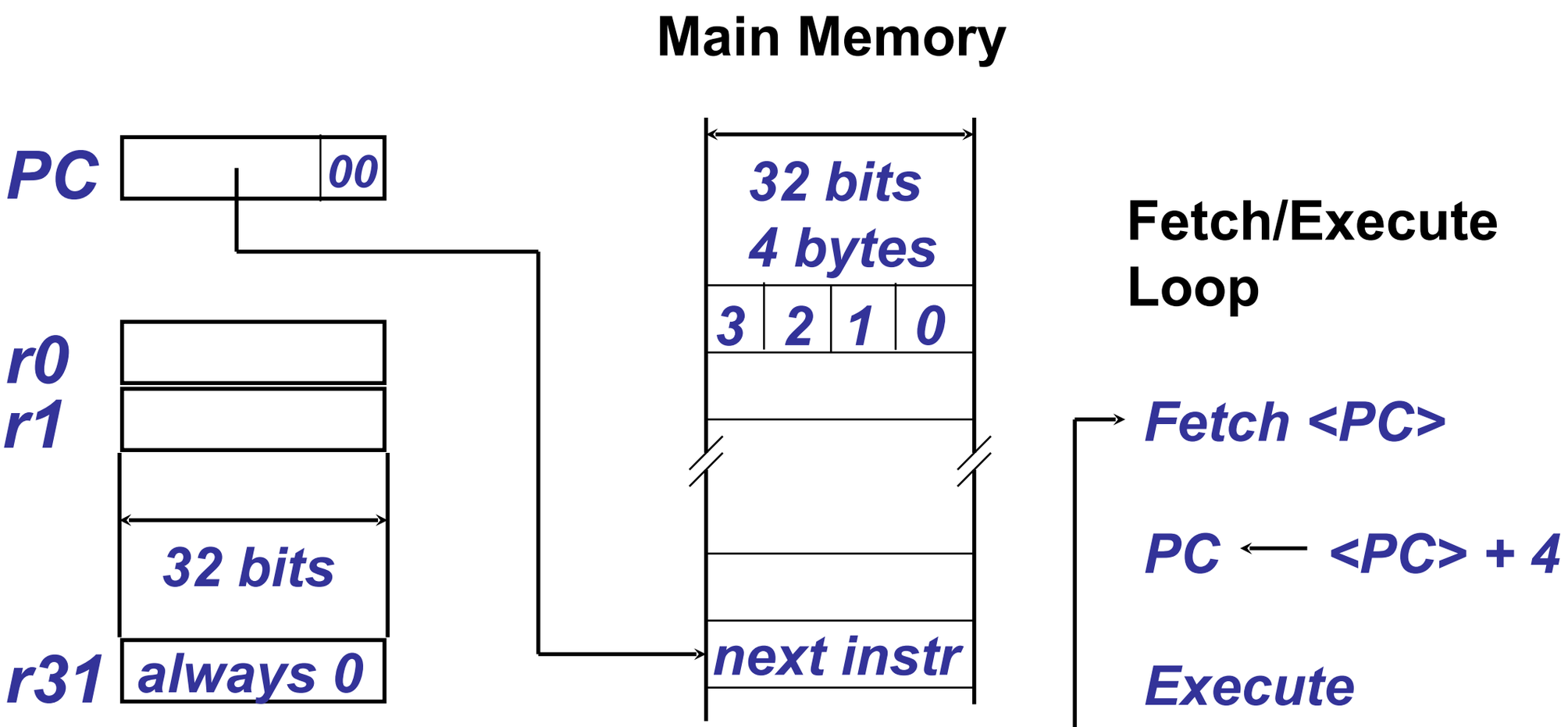
Instruction Set

- A contract or interface that specifies how you program your computer.
- Two components:
  - Visible state (registers, memory, etc.)
  - Operations on state (instructions)
- General model:



The Beta CPU

- Simplified version of the DEC Alpha designed for MIT's 6.004 class.



Examples of Beta Code

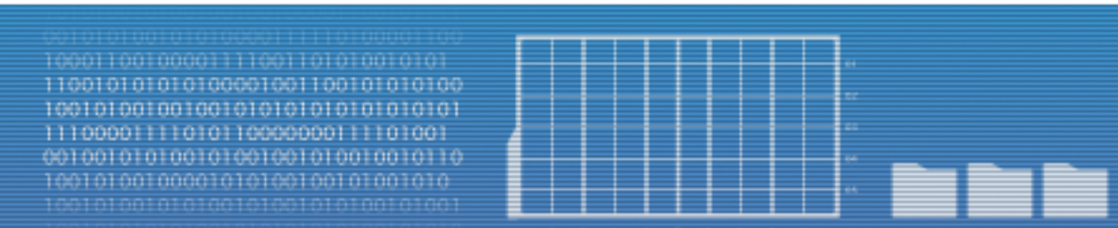
z=x+y;

```
LD(R31,X,r1)
LD(R31,Y,r2)
ADD(r1,r2,r3)
ST(r3,Z,R31)
```

if (x >= 0) y = x;

else y = -x;

```
LD(x,R0)   | R0 holds x
           | R1 will hold y
CMPLE(R31,R0,R2) | is 0<=R0?
BF(R2,elseBlock)  | no,do else
MOVE(R0,R1)
BR(done)
elseBlock:
   SUB(R31,R0,R1)
done:
    ST(R1,y)
    HALT()
.=0x200
x:    LONG(-6)
y:    LONG(0)
```



β Instructions and Formats

Form 1:

OPCODE	*rc*	*ra*	*rb*	unused
6	*5*	*5*	*5*	*11*

ALU operations with two source registers

ADD(ra, rb, rc)

SUB(ra, rb, rc)

Form 2:

OPCODE	*rc*	*ra*	literal C (signed)
6	*5*	*5*	*16*

ALU operations with one source register and constant

ADDC(ra, const, rc)

Loads and stores

LD(ra, C, rc)

ST(rc, C, ra)

Branches and Jumps

BNE(ra, label, rc)

JMP(ra, rc)

Arithmetic / Comparison Operations

ADD(ra, rb, rc) *rc* ← *<ra> + <rb>*

"Add the contents of *ra* to the contents of *rb* and store the result in *rc*".

Similarly for:
SUB
MUL
AND
OR
XOR
SHL
SHR
SRA
CMPEQ
CMPLT
CMPLE

ADDC(ra, const, rc) *rc* ← *<ra> + const*

"Add the contents of *ra* to *const* and store the result in *rc*".

16-bit signed constant

Loads and Stores

LD(ra, C, rc)

address ← <ra> + C

rc ← Memory[address]

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

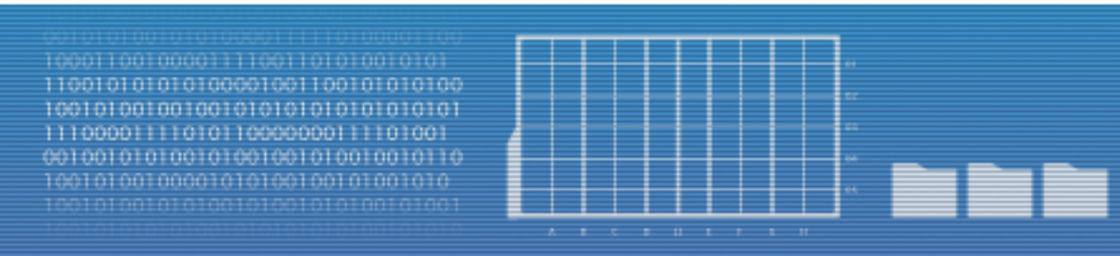
C is sign extended to 32 bits

ST(rc, C, ra)

address ← <ra> + C

Memory[address] ← <rc>

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



DISCS

Branch if nonzero

BNE(ra, label, rc) *PC* ⟵ *<PC> + 4*

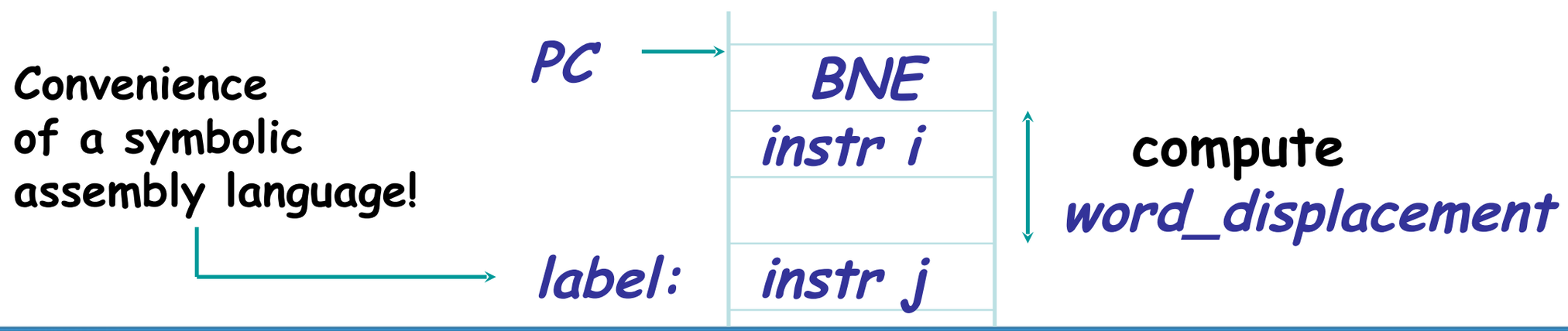
rc ⟵ *<PC>*

if <ra> nonzero then

PC ⟵ *<PC>+4*word_displacement*

aka BT "branch if True"
since CMP instructions
return non-zero if True

"Fetch into *rc* the address of the next instruction.
If ra does not contain zero then add *4* times
***word_displacement* to the value of *PC*".**



Branch if zero and Jump

BEQ(ra, label, rc)

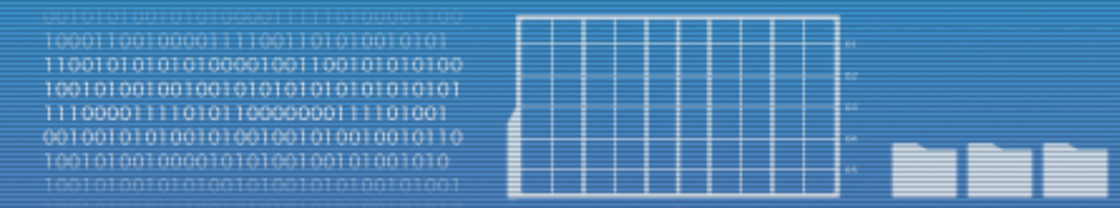
PC ⟵ <PC> + 4
rc ⟵ <PC>
if <ra> zero then
*PC ⟵ <PC>+4*word_displacement*

aka BF "branch if False" since CMP instructions return zero if False

JMP(ra, rc)

PC ⟵ <PC> + 4
rc ⟵ <PC>
PC ⟵ <ra>

"Fetch into *rc* the address of the next instruction. Load the *PC* with the address contained in *ra*".

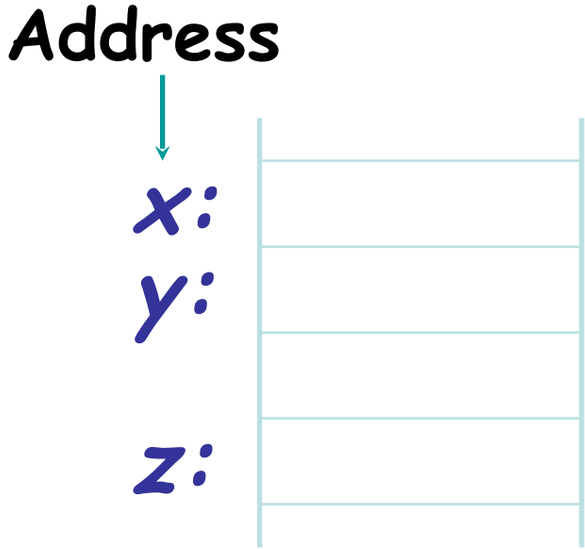


DISCS

Let's Play "Compiler"!

Rules:

Variables live in memory
Operations done in ALU
Registers hold temporary values



Statement:

```
z = (x + y) * (x - y) ;
```

```
LD(r31, x, r1)
LD(r31, y, r2)
ADD(r1, r2, r3)
SUB(r1, r2, r4)
```

Compiling Statements

Statement:

variable = expression ; → *code for expression* → *rz*

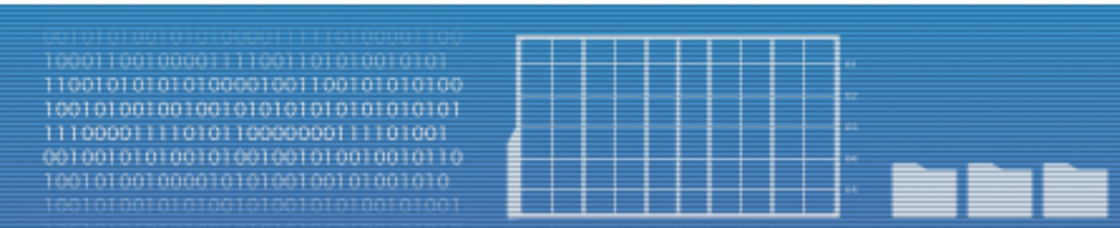
ST(rz, var_addr, r31)

expression:

exp1 OP exp2

code for exp1 → *rx*

code for exp2 → *ry*



Compiling Blocks of Statements

```
Block:
      { statement 1 ;
        statement 2 ;
              .
              .
        statement k ; }
```

code for statement 1

code for statement 2

.

.

code for statement k

Conditional Statements

```
if   expression   then
        block 1
else
        block 2 ;
```

a.k.a. predicate
= 0 implies false
≠ 0 implies true

code for expression → rz

BF(rz, elseBlock, r31)

code for block 1

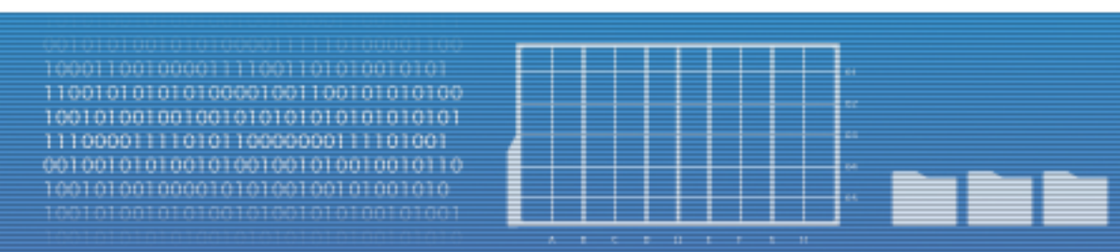
BEQ(r31, done, r31)

elseBlock:

code for block 2

always branches

done:



Iterations

while expression do
block ;

whileBlock:

code for expression → *rz*

BF(rz, done, r31)

code for block

BEQ(r31, whileBlock, r31)

done: