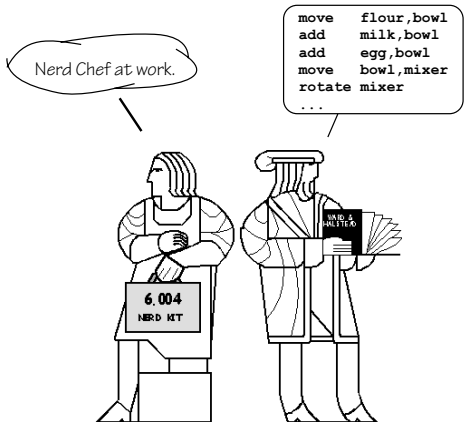
6.004 Computation Structures
Spring 2009

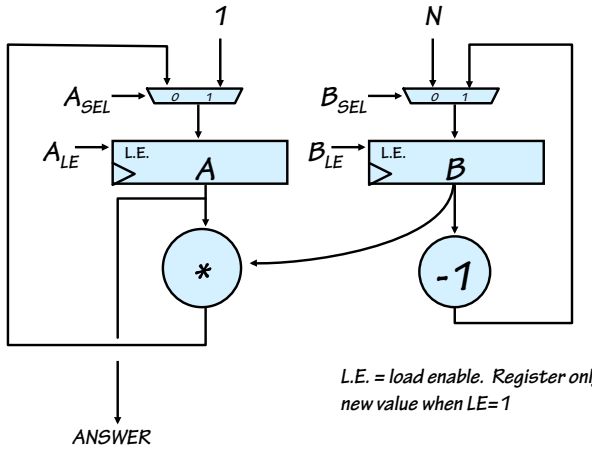
Designing an Instruction Set



Quiz 2 FRIDAY

Let's Build a Simple Computer

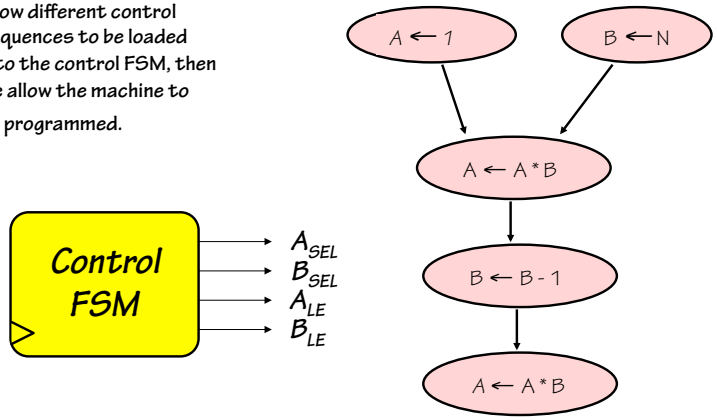
Data path for computing N*(N-1)



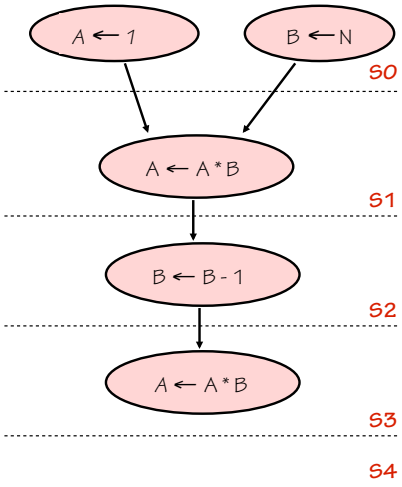
L.E. = load enable. Register only loads new value when LE=1

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.



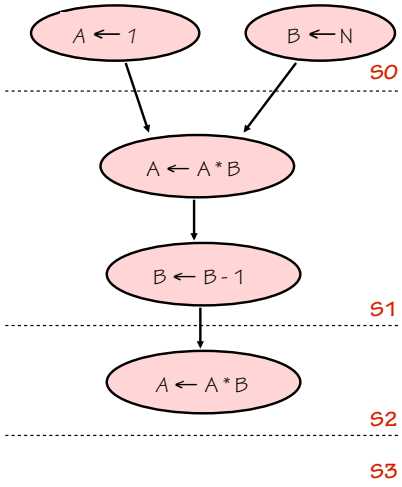
A First Program



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

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

An Optimized Program



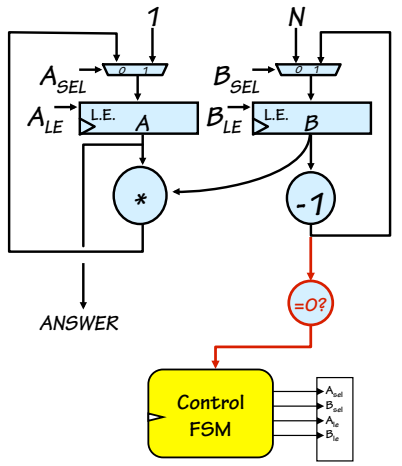
Some parts of the program can be computed simultaneously:

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

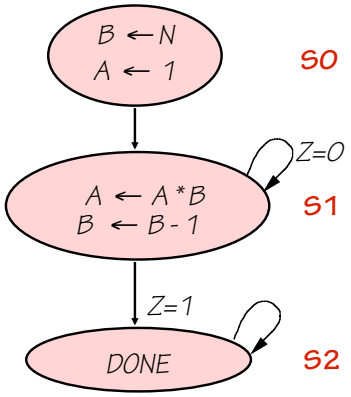
Computing Factorial

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:



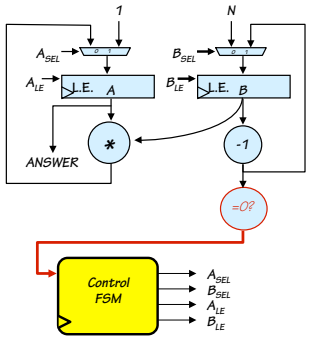
Control Structure for Factorial



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 an easier way to control it.

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

A Programmable Engine



We've used the same data paths for computing N*(N-1) and Factorial; there are a variety of other computations we might implement simply by re-programming the control FSM.

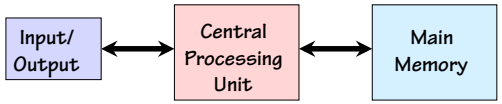
Although our little machine is programmable, it falls short of a practical general-purpose computer – and fails the Turing Universality test – for three primary reasons:

1. It has very limited storage: it lacks the "expandable" memory resource of a Turing Machine.
2. It has a tiny repertoire of operations.
3. The "program" is fixed. It lacks the power, e.g., to generate a new program and then execute it.

A General-Purpose Computer

The von Neumann Model

Many architectural approaches to the general purpose computer have been explored. The one on which nearly all modern, practical computers is based was proposed by John von Neumann in the late 1940s. Its major components are:



Ah, an FSM!

Like an Infinite Tape?

Hmm, guess J. von N was an engineer after all!

Central Processing Unit (CPU): containing several registers, as well as logic for performing a specified set of operations on their contents.

Memory: storage of N *words* of W bits each, where W is a fixed architectural parameter, and N can be expanded to meet needs.

I/O: Devices for communicating with the outside world.

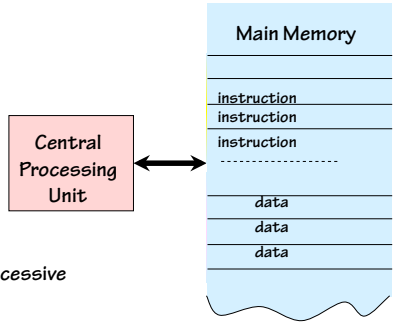
The Stored Program Computer

The von Neumann architecture easily addresses the first two limitations of our simple programmable machine example:

- A richer repertoire of operations, and
- An expandable memory.

But how does it achieve *programmability*?

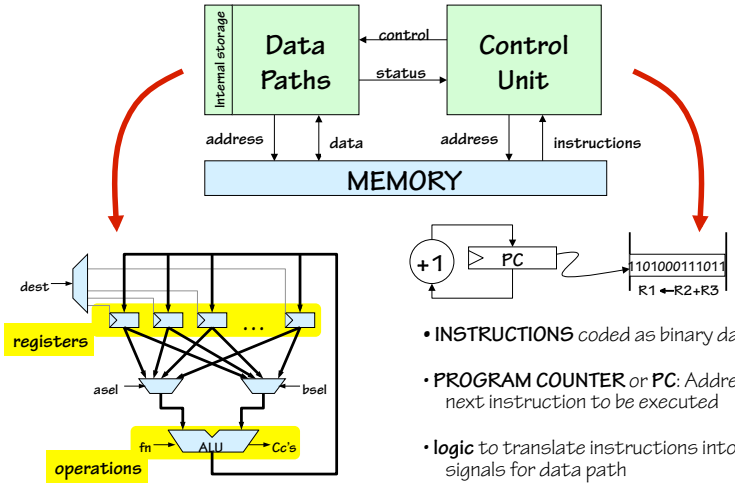
Key idea: Memory holds not only data, but *coded instructions* that make up a *program*.



CPU fetches and executes – interprets – successive instructions of the program ...

- Program is simply data for the interpreter, specifying what computation to perform
- Single expandable resource pool – main memory – constrains both data and program size.

Anatomy of a von Neumann Computer



- **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*

Instruction Set Architecture

Coding of instructions raises some interesting choices...

- Tradeoffs: performance, compactness, programmability
- Uniformity. Should different instructions
  - Be the same size?
  - Take the same amount of time to execute?
    - Trend: Uniformity. Affords simplicity, speed, pipelining.
- Complexity. How many different instructions? What level operations?
  - Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc
    - "Reduced Instruction Set Computer" (RISC) philosophy: simple instructions, optimized for speed

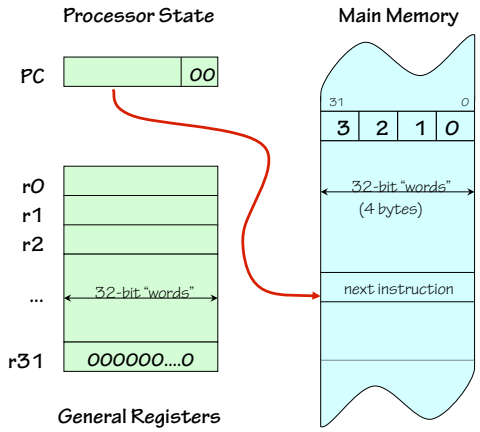
Mix of engineering & Art...

Trial (by simulation) is our best technique for making choices!

Our representative example: the β architecture!

β Programming Model

a representative, simple, contemporary RISC



Fetch/Execute loop:

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

†*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 Instruction Formats

All Beta instructions fit in a single 32-bit word, whose fields encode combinations of

- a 6-bit OPCODE (specifying one of < 64 operations)
- several 5-bit OPERAND locations, each one of the 32 registers
- an embedded 16-bit constant ("literal")

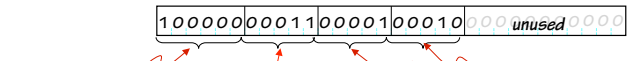
There are two instruction formats:

- Opcode, 3 register operands (2 sources, destination)

- Opcode, 2 register operands, 16-bit literal constant

β ALU Operations

Sample coded operation: ADD instruction



OPCODE = 100000, encoding ADD

r_c=3, encoding R3 as destination

r_a=1, r_b=2 encoding R1 and R2 as source locations

32-bit hex: 0x80611000

What *we* prefer to write: `ADD(r1,r2,r3)` "assembly language"

`ADD(ra,rb,rc)`:

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

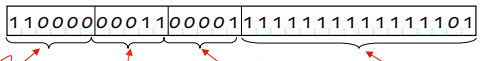
"Add the contents of ra to the contents of rb; store the result in rc"

Similar instructions for other ALU operations:

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

β ALU Operations with Constant

ADDC instruction: adds constant, register contents:



OPCODE = 110000, encoding ADDC

r_c=3, encoding R3 as destination

r_a=1, encoding R1 as first operand

constant field, encoding -3 as second operand (sign-extended!)

Symbolic version: `ADDC(r1,-3,r3)`

`ADDC(ra,const,rc)`:

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

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

Similar instructions for other ALU operations:

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

Do We Need Built-in Constants?

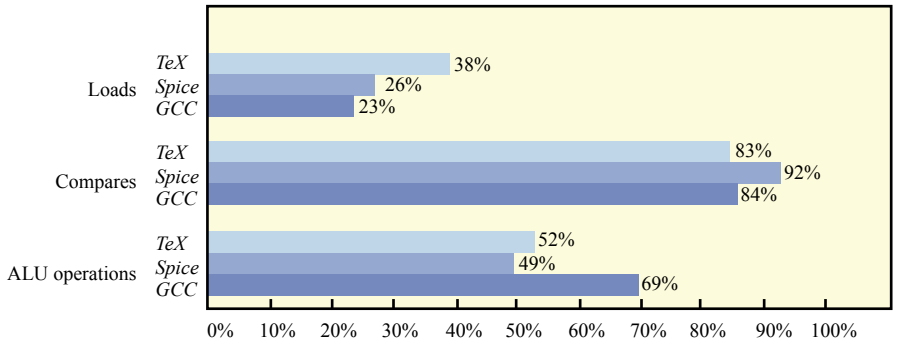


Figure by MIT OpenCourseWare.

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, …).

Baby's First Beta Program

(fragment)

Suppose we have N in r1, and want to compute N*(N-1), leaving the result in r2:

```
SUBC(r1,1,r2)    | put N-1 into r2
MUL(r2,r1,r2)    | leave N*(N-1) in r2
```

These two instructions do what our little ad-hoc machine did. Of course, limiting ourselves to registers for storage falls short of our ambitions.... it amounts to the finite storage limitations of an FSM!

Needed: instruction-set support for reading and writing locations in main memory...

β Loads & Stores

address

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"

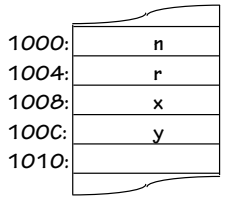
Abbreviation: `ST(rc,C)` for `ST(rc,C,R31)`

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



translates to

or, more humanely, to

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

Compilation approach: LOAD, COMPUTE, STORE

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

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

Ra defaults to R31 (0)

Common "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! Is the complexity worth the cost?
Need a cost/benefit analysis!

Memory Operands: Usage

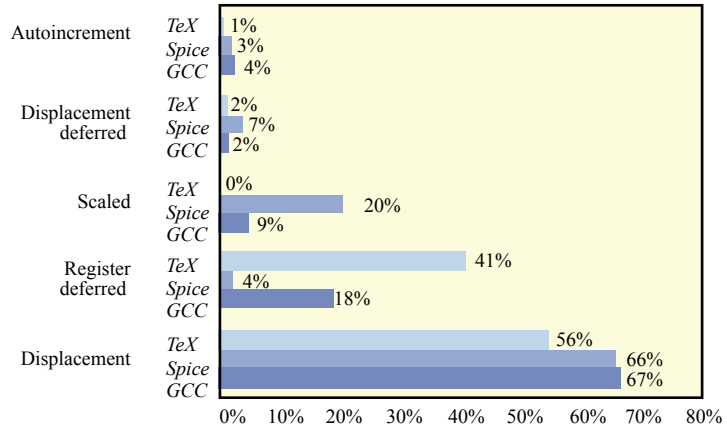


Figure by MIT OpenCourseWare.

Usage of different memory operand modes

Capability so far: Expression Evaluation

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

NB: Here we assume that variable addresses fit into 16-bit constants!

Can We Run *Every* Algorithm?

Model thus far:

- Executes instructions sequentially –
- Number of operations executed = number of instructions in our program!

Good news: programs can't "loop forever"!

- *Halting problem** is solvable for our current Beta subset!

*more next week!

NOT *Universal**

Bad news: can't compute Factorial:

- Only supports bounded-time computations;
- Can't do a loop, e.g. for Factorial!

Needed: ability to change the PC.

Beta Branch Instructions

The Beta's *branch instructions* provide a way of conditionally changing the PC to point to some nearby location...

... and, optionally, remembering (in Rc) where we came from (useful for procedure calls).

NB: "offset" is a SIGNED CONSTANT encoded as part of the instruction!

BEQ(ra,label,rc): Branch if equal

```
PC = PC + 4;
Reg[rc] = PC;
if (REG[ra] == 0)
   PC = PC + 4*offset;
```

BNE(ra,label,rc): Branch if not equal

```
PC = PC + 4;
Reg[rc] = PC;
if (REG[ra] != 0)
   PC = PC + 4*offset;
```

```
offset = (label - <addr of BNE/BEQ>)/4 - 1
       = up to 32767 instructions before/after BNE/BEQ
```


Now we can do Factorial...

Synopsis (in C):

- Input in n, output in ans
- r1, r2 used for temporaries
- follows algorithm of our earlier data paths.

```
int n, ans;
r1 = 1;
r2 = n;
while (r2 != 0) {
   r1 = r1 * r2;
   r2 = r2 - 1
}
ans = r1;
```

Beta code, in assembly language:

```
n:      long(123)
ans:    long(0)
        ...
        ADDC(r31, 1, r1)      | r1 = 1
        LD(n, r2)             | r2 = n
loop:   BEQ(r2, done, r31)     | while (r2 != 0)
        MUL(r1, r2, r1)              | r1 = r1 * r2
        SUBC(r2, 1, r2)              | r2 = r2 - 1
        BEQ(r31, loop, r31)   | Always branches!
done:   ST(r1, ans, r31)       | ans = r1
```


Summary

- Programmable data paths provide some algorithmic flexibility, just by changing control structure.
- Interesting control structure optimization questions – e.g., what operations can be done simultaneously?
- von Neumann model for general-purpose computation: need
  - support for sufficiently powerful operation repertoire
  - Expandable Memory
  - Interpreter for program stored in memory
- ISA design requires tradeoffs, usually based on benchmark results: art, engineering, evaluation & incremental optimizations
- Compilation strategy
  - runtime "discipline" for software implementation of a general class of computations
  - Typically enforced by compiler, run-time library, operating system. We'll see more of these!