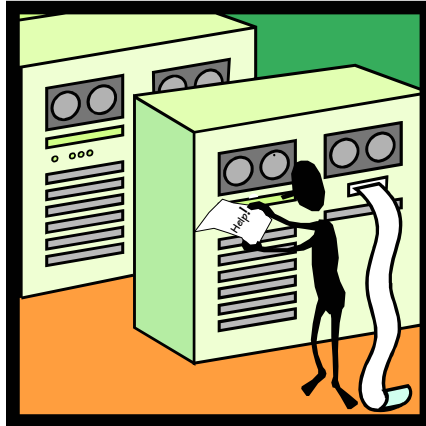


Programmable Machines



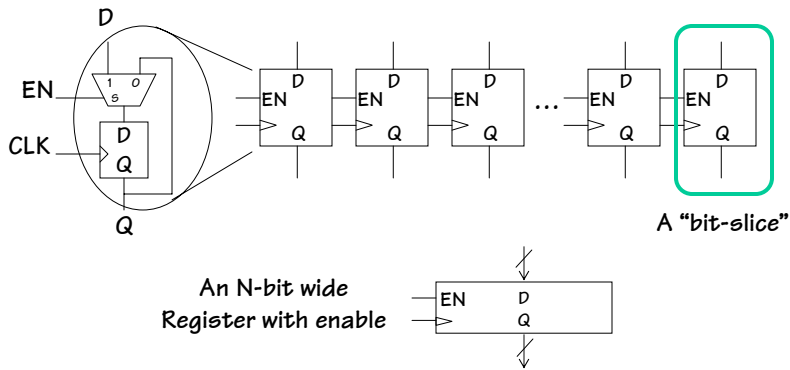
In order to compute we need only storage, data paths, and control

A General Purpose computer requires one additional thing

PROGRAMMABILITY

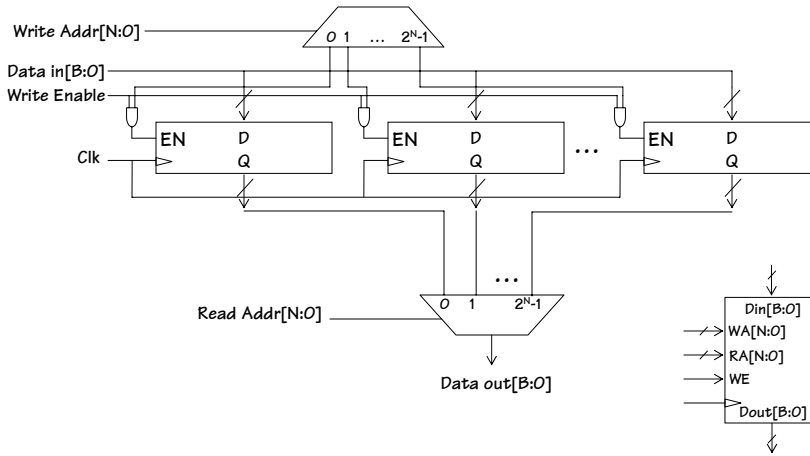
One More Functional Unit

Thus far, our processing blocks units have focused on logical and arithmetic functions. We'll also need functional units for storing intermediate results. By now, we are used to the notion of building wide registers.



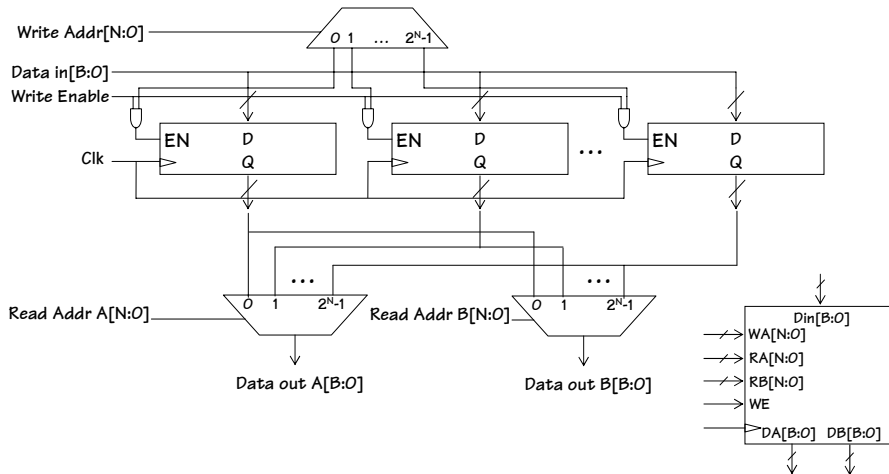
A Register File

We can also construct an addressable array of registers



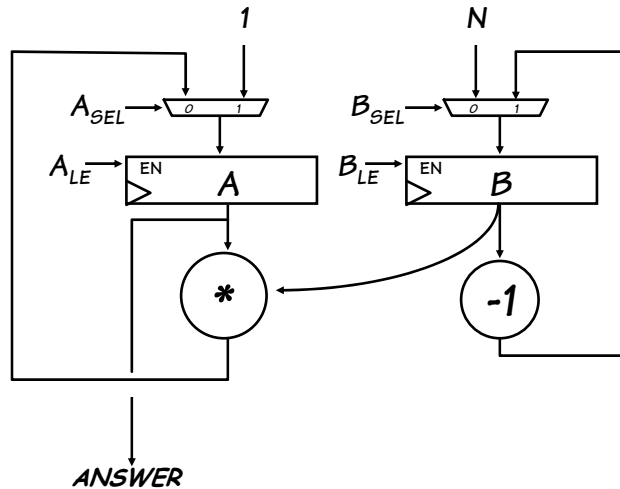
A Multi-Ported Register File

Multiple read ports by simply adding more output MUXs



Let's Build a Simple Computer

Data path for computing $N*(N-1)$



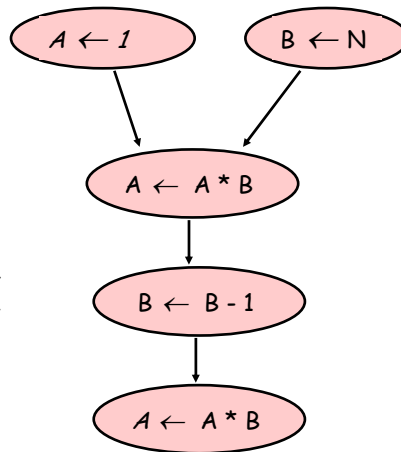
Comp 120 Spring 2005

02/22/04

L10 - Programmable Machines 5

A 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.

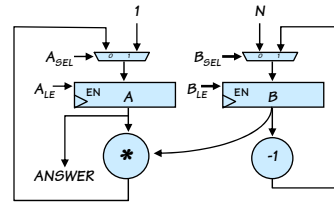
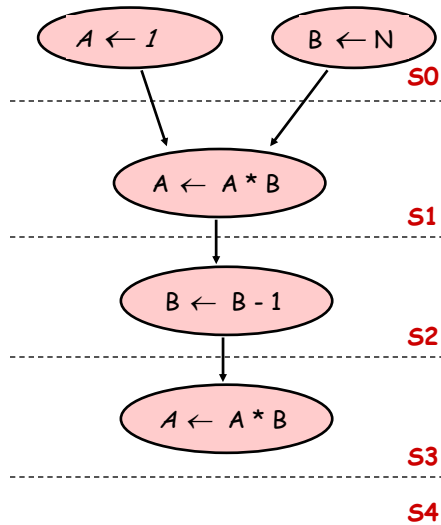


Comp 120 Spring 2005

02/22/04

L10 - Programmable Machines 6

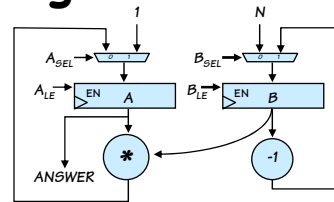
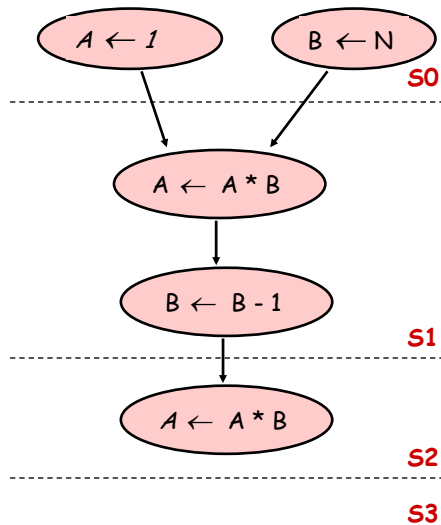
A First Program



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



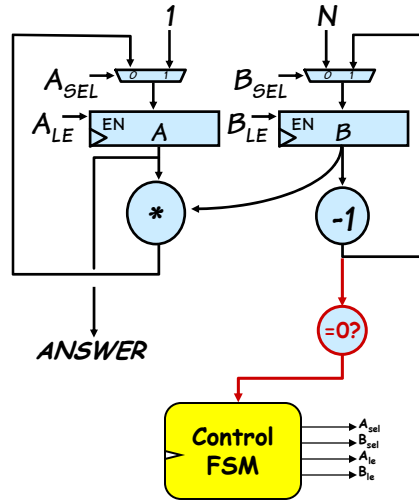
Some parts of the program can be computed simultaneously:

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
2	3	0	0	0	0
4					

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:

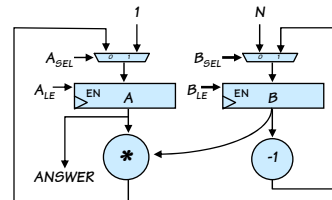
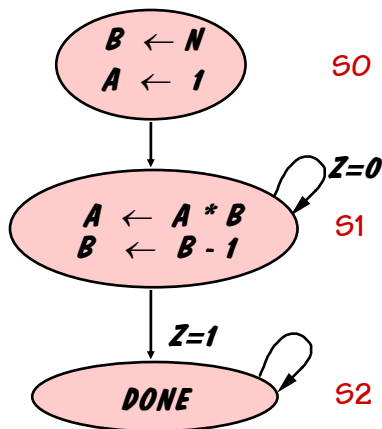


Comp 120 Spring 2005

02/22/04

L10 - Programmable Machines 9

Program for Factorial



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

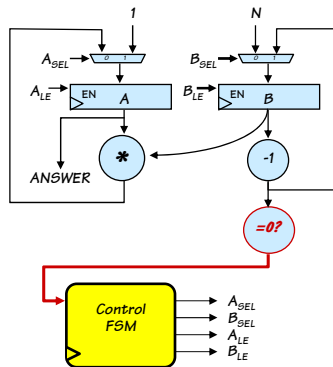
Z	S_N	S_{N+1}	A_{sel}	A_{LE}	B_{sel}	B_{LE}
-	0	1	1	1	0	1
0	1	1	0	1	1	1
1	1	2	0	1	1	1
-	2	2	0	0	0	0

Comp 120 Spring 2005

02/22/04

L10 - Programmable Machines 10

A Programmable Engine



We've used the same data paths for computing $N*(N-1)$ and Factorial, only difference: information in the control ROM.

This computing machine does a great job computing these specific functions, but is it general purpose enough?

Although our little machine is programmable, it falls short of a practical general-purpose computer – 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.

Z	S	S'	A_sel	B_sel	A_le	B_le	
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	-	-	0	0		done

Our Next Step

We need a

General Purpose Computer Architecture

- General Purpose Data path
One that can perform all the functions that we need
- General Purpose Data storage
To save all the variables and that we'll need
- ENCODED sequence of steps that can perform any function that we need... the PROGRAM!

Thus far, FSM's have been our approach to controlling the operation of data paths...

Computability vs. Programmability

Recall Church's thesis (from Lecture 7)

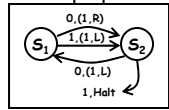
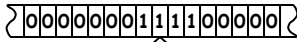
"Any discrete function computable by ANY realizable machine is computable by some Turing Machine"

And Thusly, we've defined what it means to COMPUTE (whatever a TM can compute)

A Turing machine is nothing more than an FSM that receives inputs from, and outputs onto, an infinite tape.

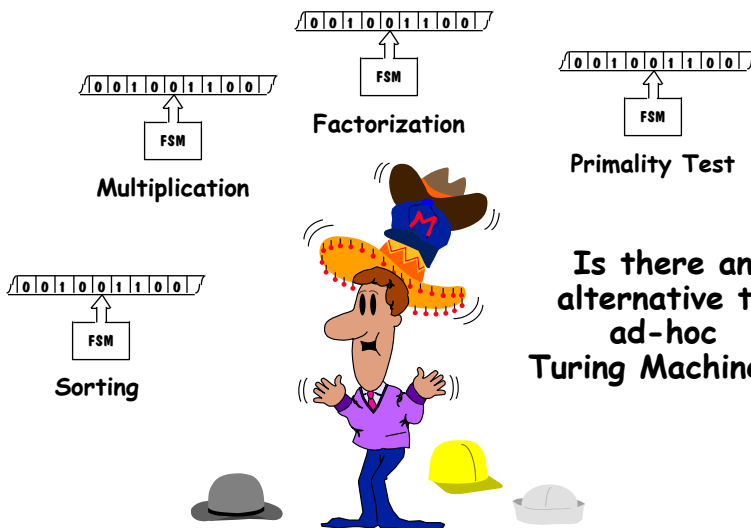
Thus far, we've been designing a new control FSMs for each new function that we encounter.

Wouldn't it be nice if we could design a more general purpose computing machine?



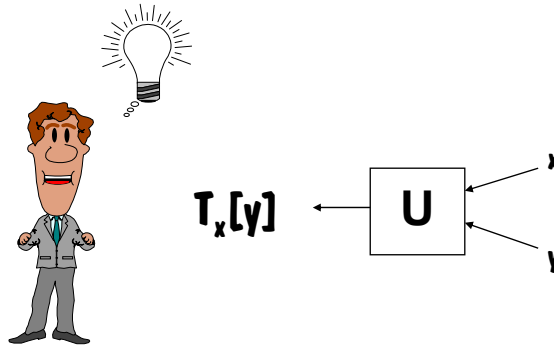
Alan Turing

Too many Turing machines!



Programs as Data

What if we encoded the description of the FSM on our tape, and then wrote a general purpose FSM to read the tape and emulate the behavior of the encoded machine? Since the FSM is just a look-up table, and our machine can make reference to it as often as it likes, it seems possible that such a machine could be built.

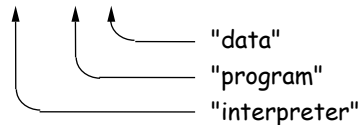


Fundamental Result: Universality

Define "Universal Function": $U(x,y) = T_x(y)$ for every $x, y \dots$
 Surprise! U is computable,
 hence $U(x,y) = T_U(\langle x,y \rangle)$ for some U .

Universal Turing Machine (UTM):

$$T_U[\langle y, z \rangle] = T_y[z]$$



PARADIGM for General-Purpose Computer!

INFINITELY many UTMs ...
 Any one of them can evaluate any computable function by simulating/emulating/interpreting the actions of Turing machine given to it as an input.

UNIVERSALITY:
 Basic requirement for a general purpose computer

Demonstrating Universality

Suppose you've designed Turing Machine T_K and want to show that it's universal.

APPROACH:

1. Find *some known* universal machine, say T_U .
2. Devise a program, P , to *simulate* T_U on T_K :
 $T_K[\langle P, x \rangle] = T_U[x]$ for all x .
3. Since $T_U[\langle y, z \rangle] = T_Y[z]$, it follows that, for all y and z .

$$T_K[\langle P, \langle y, z \rangle \rangle] = T_U[\langle y, z \rangle] = T_Y[z]$$

CONCLUSION: Armed with program P , machine T_K can mimic the behavior of an arbitrary machine T_Y operating on an arbitrary input tape z .

HENCE T_K can compute any function that can be computed by any Turing Machine.

Interpretive Layers: What's going on?

$$T_K[\langle P, \langle y, z \rangle \rangle] = T_U[\langle y, z \rangle] = T_Y[z]$$

Multiple levels of interpretation:

$T_Y[z]$	Application (Desired user function)
$T_U[\langle y, z \rangle]$	Portable Language / Virtual Machine
$T_K[\langle P, \langle y, z \rangle \rangle]$	Computing Hardware / Bare Metal

Benefits of Interpretation:

BOOTSTRAP high-level functionality on very simple hardware.

Deal with “**IDEAL**” machines rather than real machines.

REAL MACHINES are built this way - several interpretive layers.

Power of Interpretation

BIG IDEA: Manipulate *coded representations* of computing machines, rather than the machines themselves.

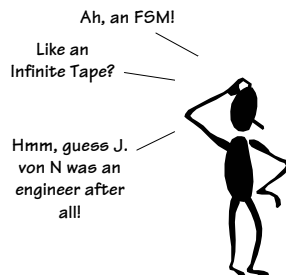
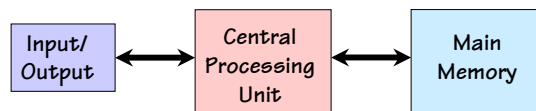
- PROGRAM as a behavioral description
- SOFTWARE vs. HARDWARE
- INTERPRETER as machine which takes program and mimics behavior it describes
- LANGUAGE as interface between interpreter and program
- COMPILER as translator between languages:

INTELLECTUAL BENEFITS:

- Programs as data -- mathematical objects
- Combination, composition, generation, parameterization, etc.

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:



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.

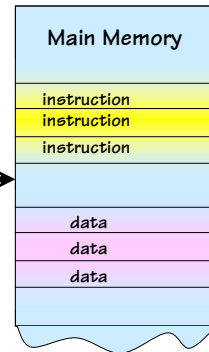
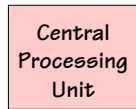
The Stored Program Computer

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?

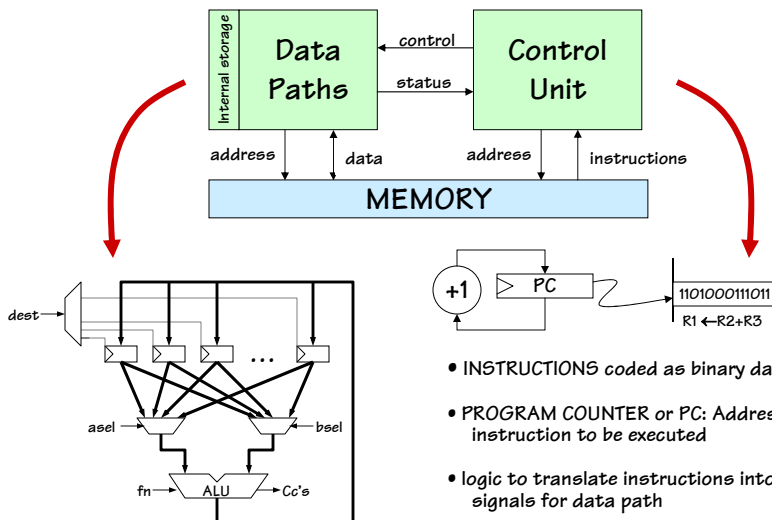
Key idea: Memory holds not only data, but coded instructions that make up a program.



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

- Program is simply data for the interpreter – as in a Universal Turing Machine!
- Single expandable resource pool – main memory – constrains both data and program size.

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

Lingering Questions:

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:

- what instructions do we need?
- 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	beel
------	----	------	------	------