EECS 314 Computer Architecture
The first operational stored-program computer
& RISC Project
Instructor: Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University
This presentation uses powerpoint animation: please viewshow

EDSAC 1949: the first computer

Designed and built at Cambridge University, England, the EDSAC is the first full-scale *operational stored-program computer,* and is therefore the final candidate for the title of "the first computer".

The EDSAC performed its first calculation on May 6, 1949, when a length of perforated paper tape was threaded through the tape reader connected to the machine, and a few seconds later, the computer's printer began clattering out a list of numbers: 1, 4, 9, 16, 25, 36....

EDSAC Simulator: http://www.dcs.warwick.ac.uk/~edsac and Ref: http://hoc.co.umist.ac.uk/storylines/compdev/electronic/edsac.html

EDSAC: subroutines, relocatable, BIOS

• Indeed, EDSAC could access a library of programs called (would-you-believe) *subroutines,*

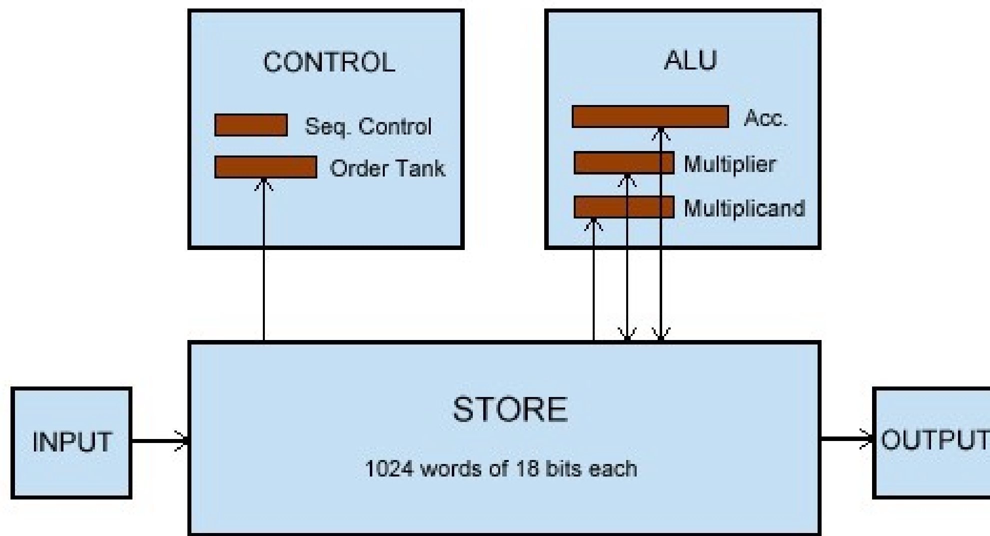
• including what was thought impossible at the time: a subroutine for numerical integration which (by calling an "auxiliary" subroutine) could be written without knowledge of the function to be integrated! (pass the *by address* of another function to a subroutine)

• A problem: whenever a tape was read the subroutine may not go to the same memory locations so certain memory addresses had to be changed. This problem was overcome by preceding each piece of code with a set of "coordinating orders", making it *self-relocatable.*

• The next major advance demonstrated by this machine, was a continuation of EDSAC's subroutine idea. The concept of a *bootstrap was invented - a program that is run every time the machine is turned on*. Today, we call that shadow ROM BIOS.

EDSAC Simulator: http://www.dcs.warwick.ac.uk/~edsac and Ref: http://hoc.co.umist.ac.uk/storylines/compdev/electronic/edsac.html

EDSAC architecture



Typical execution times were

1.5 milliseconds for the simple commands = 667 adds/sec

4.5 milliseconds for a multiply = 222 mults/sec

http://www.cl.cam.ac.uk/UoCCL/misc/EDSAC99/simulators/echo/refindex.html

EDSAC memory

Its main memory is of a type that had existed for some years, but had not been used for a computing machine: the "ultrasonic delay line" memory.

It had been invented originally by William Shockley of Bell Labs (also one of the co-inventors of the transistor, in 1948), and Presper Eckert had made an improved version in connection with radar systems.

The "delay storage" referred to an electromechanical delay line: oscillating quartz crystals generated pulses in tubes of mercury and the pulses were recycled to provide memory.

In place of mercury, Turing suggested gin and tonic because the speed of propagation was relatively insensitive to temperature changes!

Memory Store: Mercury Delay Tanks

http://kbs.cs.tu-berlin.de/~jutta/time/msb-chronology-of-dcm.html
http://home.golden.net/~pjponzo/CSH.htm

EDSAC memory: FIFOs

Quartz crystal

Hg

Gate

Reshaper

Amplifier

Write line

Control line

Read line

	READ	WRITE	RECIRCULATE
READ SELECT	1	0 OR 1	0 OR 1
WRITE SELECT	0 OR 1	1	0
DATA INPUT	0 OR 1	0 OR 1	0 OR 1
R/W ENABLE	1	1	0 OR 1

RECIRCULATE PATH

INVERT

AND

OR

DRIVER

DELAY LINE

AMPLI-FIER

DATA INPUT

AND

DATA OUTPUT

WRITE SELECT

R/W ENABLE

READ SELECT

FIG. 2.1.1. Delay line memory block diagram.

http://www.science.uva.nl/faculteit/museum/delayline.html

Memory Store: Mercury Delay Tanks

EDSAC Description

System Clock: 0.5 Mhz

Arithmetic: No overflow or carry bit. Serial +, –, × and &

Registers: A=71 bits, multiplier H=35 bits, PC=10 bits, IR=15bits.

Better than a 32 bit processor!

One Instruction format: $Opcode_{18..14}$ $Spare_{13}$ $Address_{12..2}$ $Length_{1}$

Input/Output Paper tape, Printer, 0-9 telephone dial, 16x36 video

Memory organization: 1024 words (i.e. about 2 kilobytes)

= 32 mercury tanks containing 32 18-bit words

Boot strap loader: Hardwired circuit fills first tank with 31 instructions

Today, we call that shadow ROM BIOS

Short word: Mem[n] $=Mem[n]_{18..1}$ (Bit 0 is always lost, can only use 17 bits)

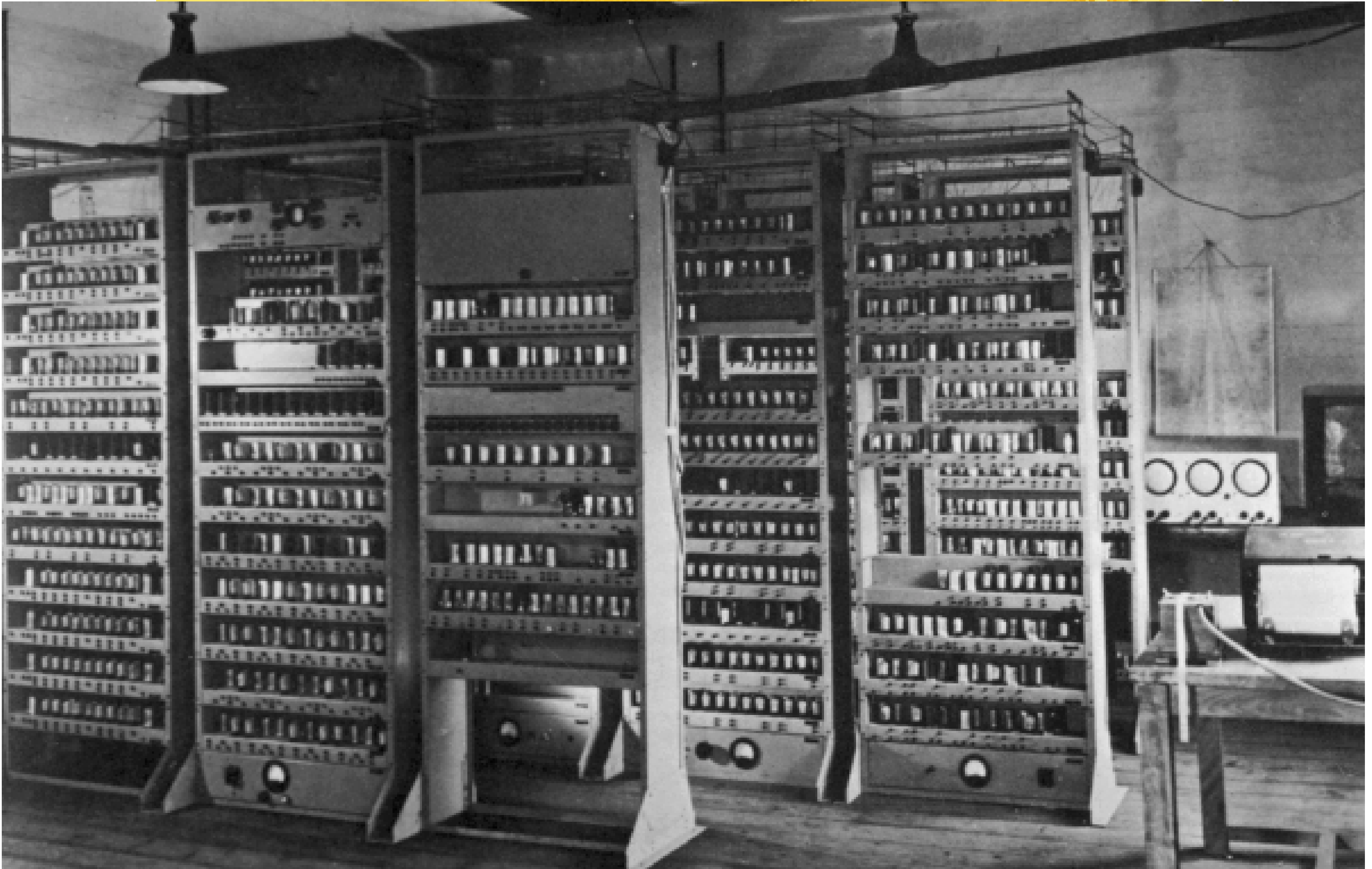
Long word: $Mem_{35..1}[n+1]$ $= Mem[n+1]_{18..0} || Mem[n]_{18..1}$

Serial Memory: can run two adjacent memory location together

Technology: 3500 Tubes

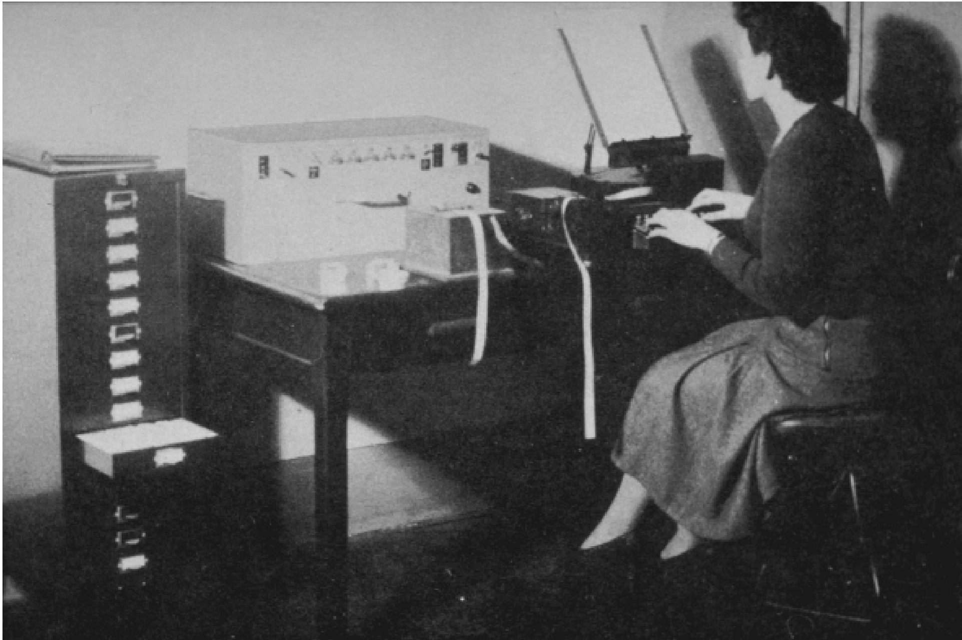
Ref: The Origins of Digital Computers, Brian Randell, 1975, 2nd, Springer-Verlag

EDSAC CPU



Ref: http://www.dcs.warwick.ac.uk/~edsac

EDSAC I/O



The University Mathematical Laboratory, Cambridge
May, 1950.

P.J.Farmer L.Foreman S.A.Barton G.J.Stevens R.Kimpton S.Gill P.Chamberlain
D.W.Willis K.N.Dodd M.Ellison B.P.Vernon H.Fye C.M.Beech R.B.Bonham-Carter A.E.Glennie D.J.Wheeler
E.E.C.McKee J.M.Bennett W.Renwick M.V.Wilkes E.N.Mutch R.A.Brooker C.M.Mumford

(Absent — B.M.Worsley D.G.N.Hunter)

EDSAC Instructions (formally called *orders*)

Instruction

A n S	$A_{70..0} = A_{70..0} + Mem[n]_{18..1} \| \| 0_{52..0}$
A n L	$A_{70..0} = A_{70..0} + Mem[n+1]_{35..1} \| \| 0_{35..0}$
A n w	$A_{70..0} = A_{70..0} + Mem.w[n]$
S n w	$A_{70..0} = A_{70..0} - Mem.w[n]$
R n S	$A_{70..0} = A_{70..0} >> n$
L n S	$A_{70..0} = A_{70..0} << n$
C n w	$A_{70..0} = A_{70..0} \& Mem.w[n]$
H n w	$H_{34..0} = Mem.w[n]$
V n w	$A_{70..0} = A_{70..0} + H_{34..0} * Mem.w[n]$
N n S	$A_{70..0} = A_{70..0} - H_{34..0} * Mem.w[n]$

EDSAC Instructions (i.e. orders)

Instruction

T n S	$Mem[n]_{18..1} = A_{70..53}; \ A_{70..0}=0;$
T n L	$Mem[n+1]_{35..1} = A_{70..36}; \ A_{70..0}=0;$
U n S	$Mem[n]_{18..1} = A_{70..53}$
U n L	$Mem[n+1]_{35..1} = A_{70..36};$
E n S	$PC_{9..0} = (A >= 0)? \ n : \ PC_{9..0}+1;$
G n S	$PC_{9..0} = (A < 0)? \ n : \ PC_{9..0}+1;$
Z S	Stop the machine and ring the warning bell
I n S	$Mem[n]_{18..14}$ = Paper Tape Reader
O n S	Printer = $Mem[n]_{18..14}$ (print character in opcode position)
F n S	$Mem[n]_{18..14}$ = Printer character buffer

EDSAC 1952 Tic-Tac-Toe program

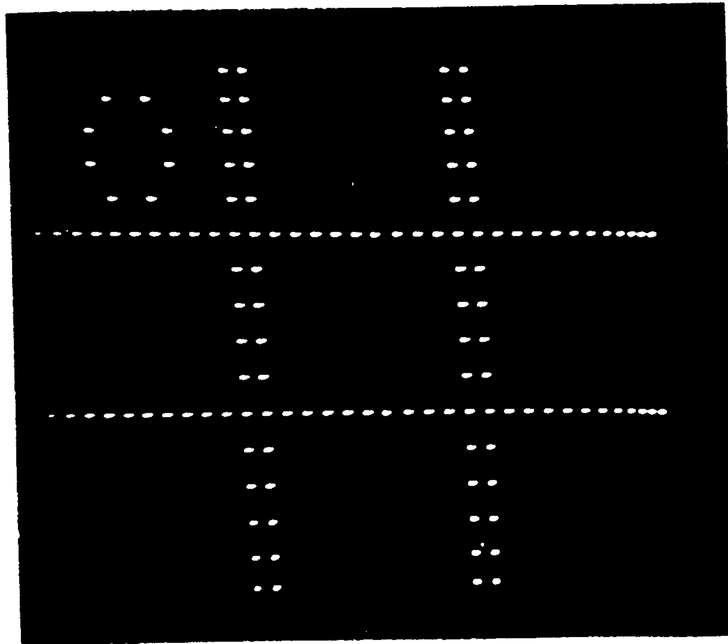
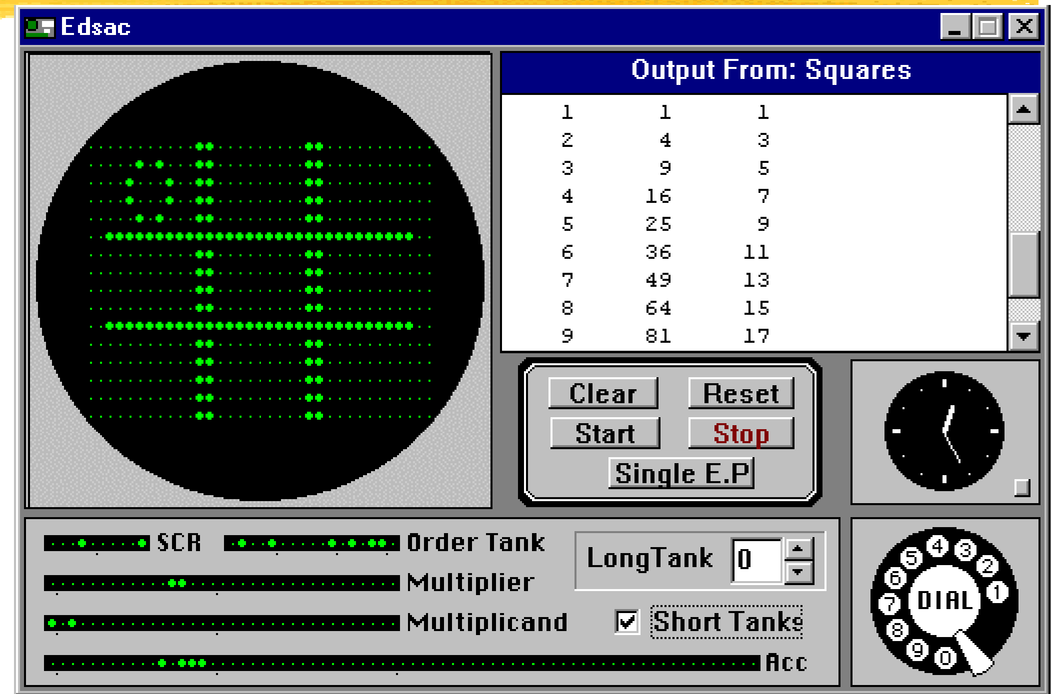


Diagram 1 (b). The board with a nought in position 9.



16 by 36 memory mapped monochrome (1-bit) video

Each memory bit corresponds to a pixel (picture element) on the display

The EDSAC Simulator: http://www.dcs.warwick.ac.uk/~edsac

Ref: http://www.cl.cam.ac.uk/UoCCL/misc/EDSAC99/

EDSAC instruction comparison

Modern computers provide instructions for

call: jal address

return: jr $ra

indexing: lw $rt, $offset($rs)

The EDVAC achieved this through *self modifying code*

At the time, the Von Neuman architecture was view as vital

(i.e. instructions and data are contained in the same memory)

For example: suppose loads on the MIPS *could not add* a base register

How would we do: lw $3,offset($1)

```
32:  addi  $2,$1,offset   #add offset plus base
36:  sh    $2,42($0)      #store within lw instruction
40:  lw    $3,0($0)
```

EDSAC Hello, World

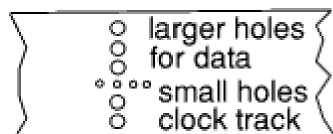
```
31:   T53S   # A=0; last line of code +1 for loader
32:   O41S   # Printer = Mem[41..52]
33:   A32S   # A=A+Mem[32]; get instruction at 32
34:   A39S   # A=A+2; add 1 to address field
35:   U32S   # Mem[32]=A; store new instruction
36:   S40S   # A=A-"O53S"; stop output?
37:   G31S   # if (A<0) then no and goto 31
38:   ZS     # stop machine and ring the bell
39:   P1S    # use instruction to define word =2
40:   O53S   # use instr. to compare last index
```

```
41: *S #letter shift
42: HS
43: ES
44: LS
45: LS
46: OS
47: !S  #blank
48: WS
49: OS
50: RS
51: LS
52: DS
```

Note that the letter code and opcode as the same

Simplifies loader (loader acted as an assembler too!)

11100 = 'A' = Add opcode



Note that the letter code and opcode as the same

Actual paper tape source input (load for initial orders 1)

T53SO41SA32SA39SU32SS40SG31SZSP1SO53S
*SHSESLSLSOS!SWSOSRSLSDS

EDSAC versus the EDVAC: battle of being the first

Before von Neumann, computer programs were stored either mechanically (on cards or even by wires that connected a matrix of points together in a special pattern like ENIAC) or in separate memories from the data used by the program.

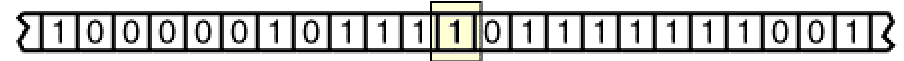
Von Neumann introduced the concept of the *stored program*—both the program that specifies what operations are to be carried out and the data used by the program are stored in the same memory.

Although EDVAC is generally regarded as the first stored program computer, Randell states that this is not strictly true [Randell94]. EDVAC did indeed store data and instructions in the same memory, but data and instructions did not have a common format and were not interchangeable.

Sadly, EDVAC was not a great success in practical terms. Its construction was (largely) completed by April 1949, *but it did not run its first applications program until October 1951.* (EDSAC was 1949)

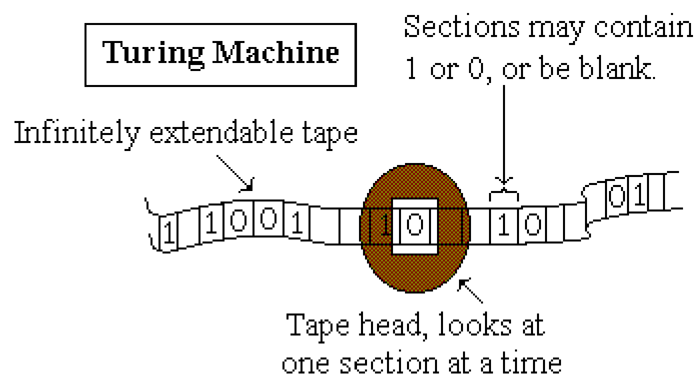
Ref: http://wheelie.tees.ac.uk/users/a.clements/History/History.htm

Turing machine



A *Turing machine* (TM) typically works as follows:

1. Read the input symbol from the tape.
2. Choose the next operation found in the state transition table (i.e. FSM), based upon the current state, and the input symbol.
3. Write the output symbol indicated in the matrix cell.
4. Transform into the next state indicated in the matrix cell.
5. Move the tape pointer in the direction indicated in the matrix cell.
6. If the next state is not H, the Halt state, start the instruction loop at the top.

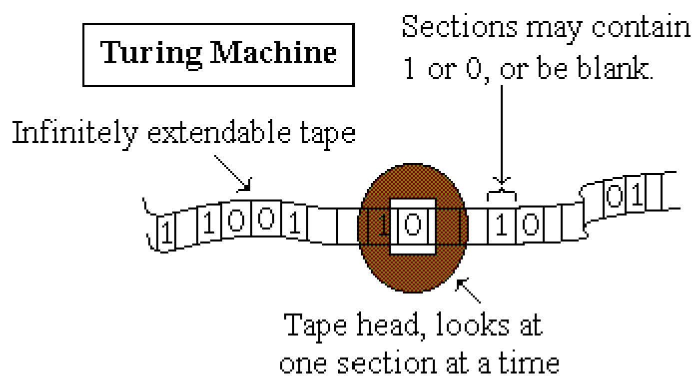


State	Read	Write	Move	Next State
S1	0	0	L	S1
	blank	1	L	S2
	1	blank	R	S1
S2	0	1	R	S2
	blank	0	R	S2
	1	1	L	S1

State Transition Table for a Turing Machine

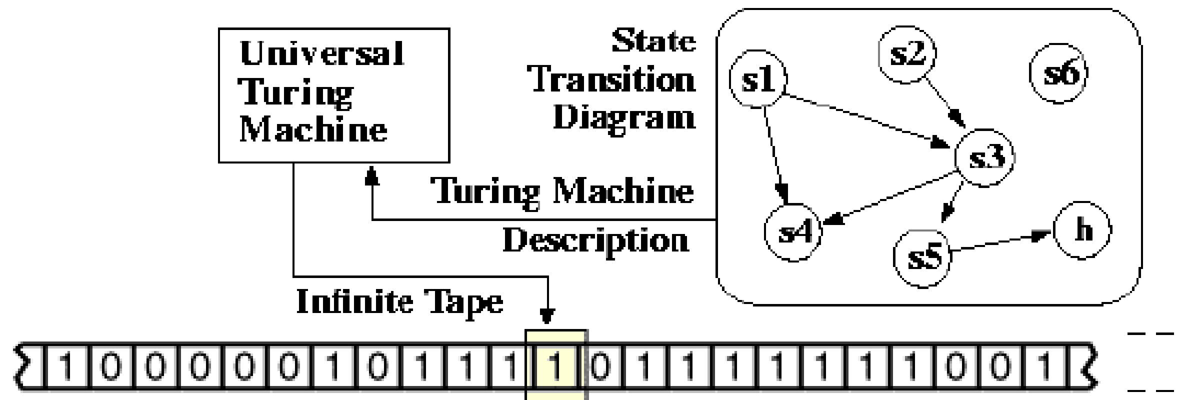
EDSAC versus the Turing machine

A Turing machine is a very simple machine, but, logically speaking, has all the power of any digital computer. It may be described as follows: A Turing machine processes an infinite tape *whereas a digital computer processes a finite tape.*



State	Read	Write	Move	Next State
S1	0	0	L	S1
	blank	1	L	S2
	1	blank	R	S1
S2	0	1	R	S2
	blank	0	R	S2
	1	1	L	S1

State Transition Table for a Turing Machine



EDVAC architecture comparison

EDVAC differs from the modern computers of today:

CPU: Serial ALU to parallel & multiple ALUs and pipelining

Registers: Serial 71 bit accumulator to 64bit parallel & multiple registers

Memory: Serial Mercury Delay Tubes to parallel DRAM CMOS

Single-level memory to multi-level: Disk, RAM, L2, L1 cache

Input: Paper tape to keyboards, mouse, scanners, cdroms, …

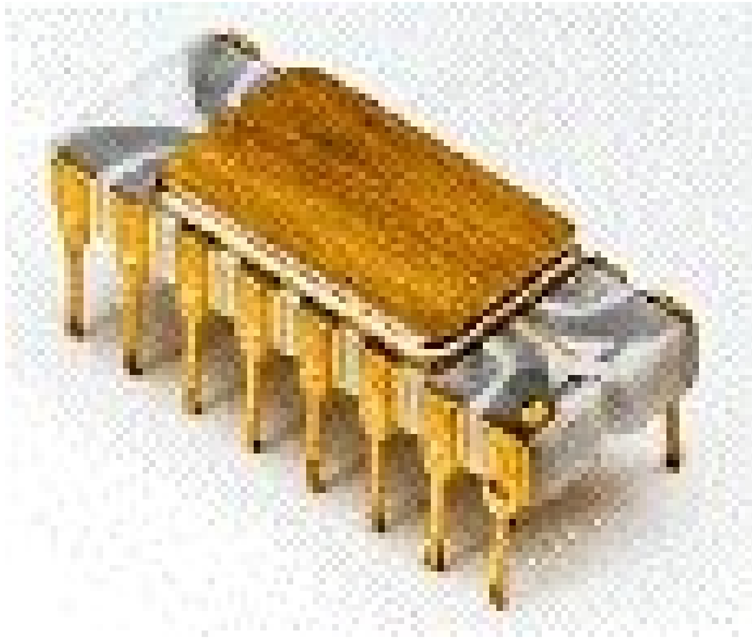
Output: Teletype printer and a bell to 24-bit video, 16-bit sound,

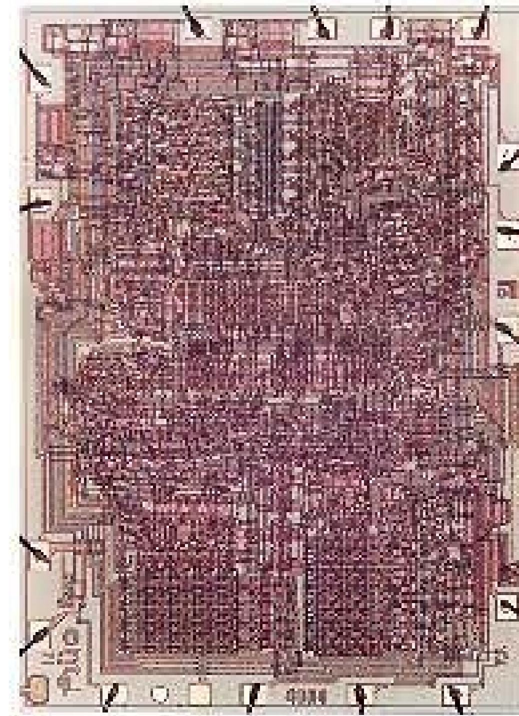
The key design components

parallelism:	achieved though architecture
switching delay:	achieved through technology (silicon)
area:	vacuum tubes to silicon
power:	vacuum tubes to silicon
cost:	mass manufacturing, marketing & sales

Intel Microprocessor History: 4004

- 1971 Intel 4004, 4-bit, 0.74 Mhz, 16 pins, 2250 Transistors



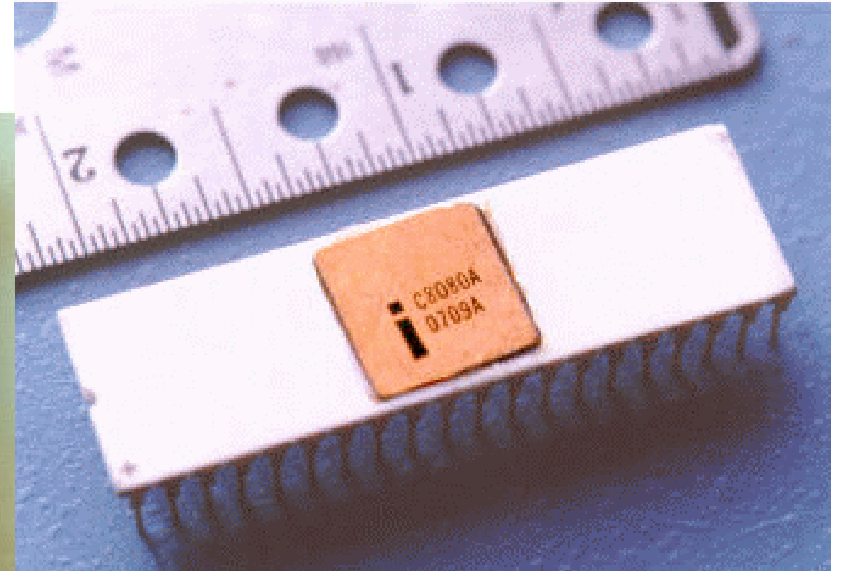


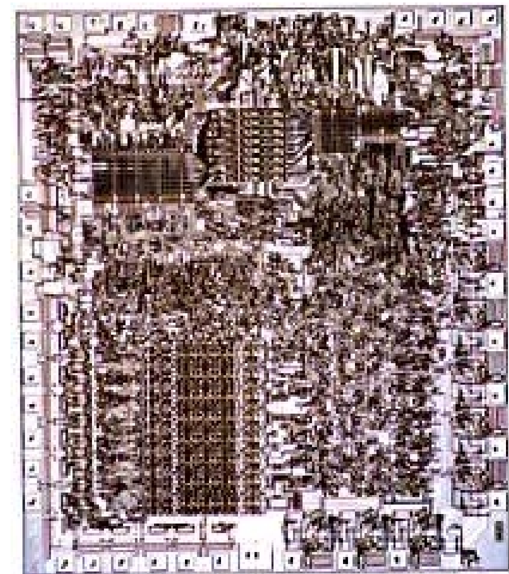
- Intel publicly introduced the world's first single chip microprocessor: U. S. Patent #3,821,715.
- Intel took the integrated circuit one step further, by placing CPU, registers, memory access, I/O on a single chip

Intel Microprocessor History: 8080

- 1974 Intel 8080, 8-bit, 2 Mhz, 40 pins, 4500 Transistors

Altair 8800 Computer



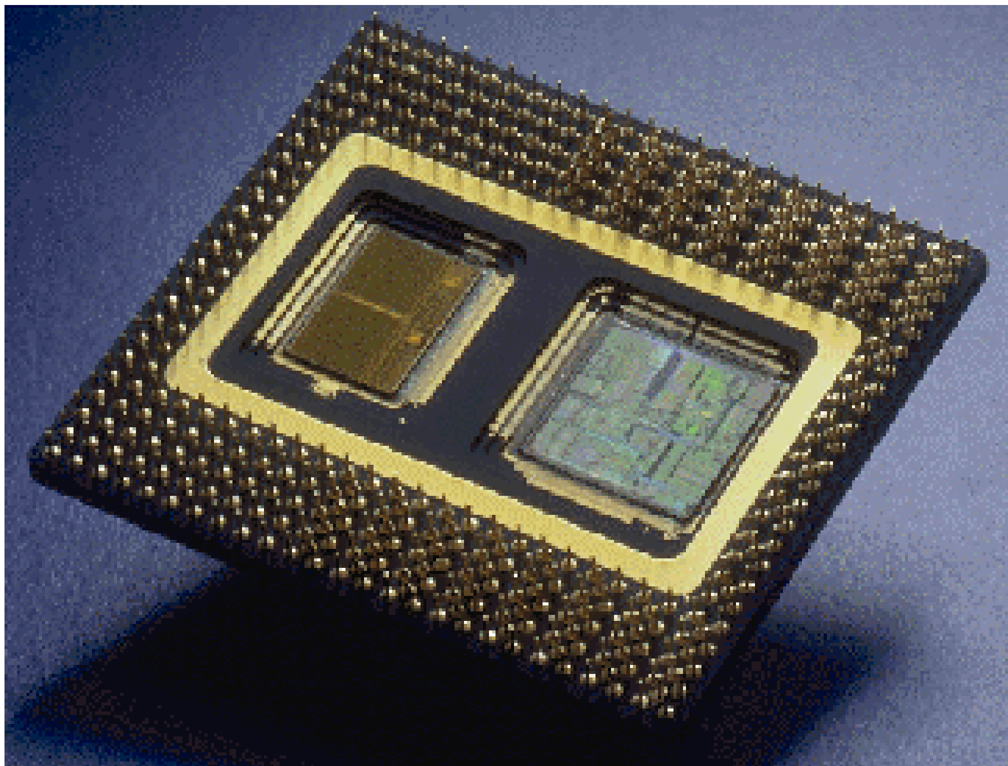


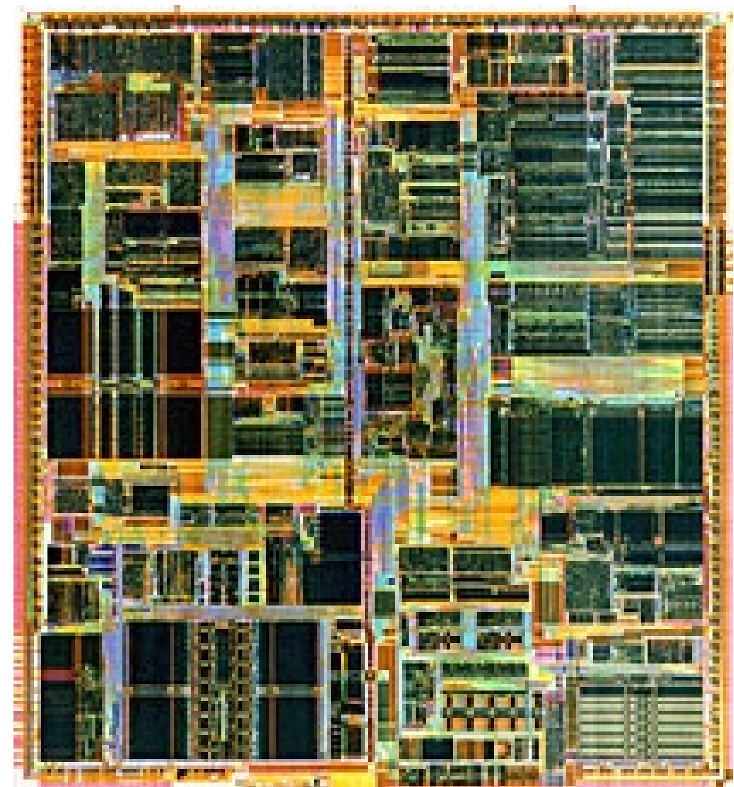
Bill Gates & Paul Allen

write their first Microsoft software product: Basic

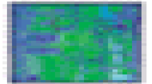
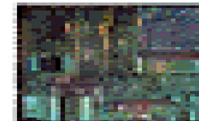
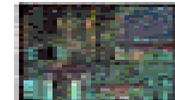
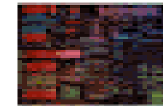
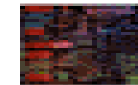
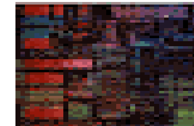
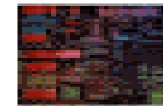
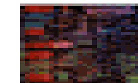
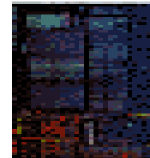
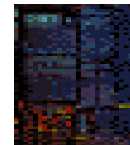
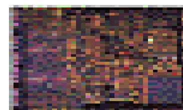
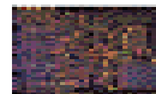
Intel Processor History: Penitum Pro

- 1995 Intel Pentium Pro, 32-bit ,200 Mhz internal clock, 66 Mhz external, Superpipelining, 16Kb L1 cache, 256Kb L2 cache, 387 pins, 5.5 Million Transistors





Intel's microprocessor evolution

silicon process technology	1.5µ	1.0µ	0.8µ	0.6µ	0.35µ	0.25µ
Intel® Pentium® III processors						
Pentium® II processors						
Pentium® Pro processor						
Pentium® processor						
Intel486™ DX processor						
Intel386™ DX processor						

RISC Project

Each team must turn in a report which contains the following

(1) Cover sheet with up to 3 team members names & signatures

(2) Description of the problem, enhancements, & lessons learned.

(3) (a) Comment "# C source code statements" followed by MIPS assembler source related it. (b) Also, comment each "# assembler source" statement. (c) Must use at least the given functions & data structure described later.

(4) Flowchart of the function: game_move( )

(5) Floppy disk of the (1)-(3).

(6) Demo with all members present with TA asking questions.

Note: you will get no credit by just handing in C code!

Wopr: example

How the program should work

```
wopr

Shall we play a game?

Global thermonuclear War

Wouldn't you prefer a
 good game of toe-tac-tic?

toe-tAc-Tic

X: please enter your move?

1

   X |   |
  ---+---+---
     | O |
  ---+---+---
     |   |
```

Strcasecmp()
Case insensitive string matching

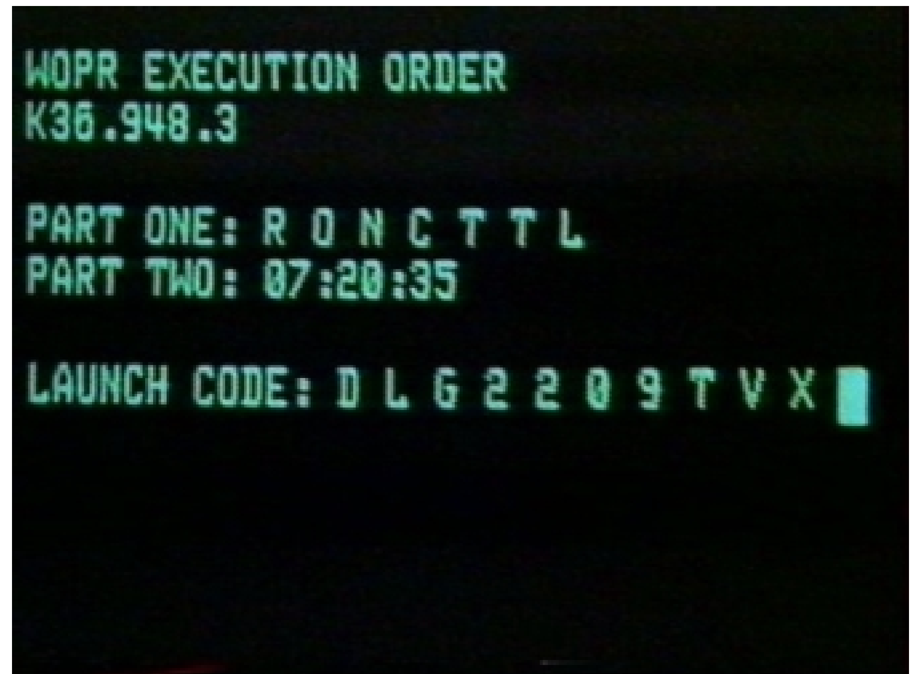
Wopr: con’t

```
X: please enter your move?
7

 x |   |
---+---+---
 o | o |
---+---+---
 x |   |

X: please enter your move?
6

 x | o |
---+---+---
 o | o | x
---+---+---
 x |   |
```



Wopr: con't

X: please enter your move?

```
 X | O |
---+---+---
 O | O | X
---+---+---
 X | X | O
```

Draw. Game over.

Shall we play a game?

List Games

1.) Toe-Tac-Tic

2.) logoff.

Shall we play a game?

logoff

logoff.

Wopr: Reverse Tic-Tac-Toe

RISC Project:

wopr: this program is inspired by the movie, wargames.

Toe-tac-tic: Reverse Tic-Tac-Toe

Object of the game:

Avoid getting three marks in a row (the opposite of tic tac toe)

The play stops when a player gets 3 in a row (loses) or a draw.

For example see: http://tictactoe.javagamz.com/toetactic.html

Wopr: functions

(see Appendix A & A-22)

Write at least these functions (using MIPS register conventions):

```
main()
```
 # Main program: reads keyboard for “logoff”, “list games”, “toe-tic-tac” and calls TICTACTOE;

```
void game_print(struct TICTACTOE *game);
```
 # prints the tic-tac-toe board (player: 1=O, 2=X, 0=blank)
 # also prints status only if win or draw

```
void game_init(struct TICTACTOE *game);
```
 # initializes the data structure board to blank

```
int game_set(struct TICTACTOE *game, position);
```
 # sets & checks for valid move for current player

```
void game_move(struct TICTACTOE *game);
```
 # generates the computers move for current player

```
int game_check(struct TICTACTOE *game);
```
 # test and sets the game status flag to draw or win
 # return 1 if game over and return to main(); else return 0

Wopr: data structure

```
struct TICTACTOE {
        signed char *board;
        short  current_player; /* 1=O, 2=X */
        short  status;
        /* -1=pending,0=draw,1=player wins,2=player wins */
};

...

game_toetictac() {
        struct TICTACTOE     toetactic;
        struct TICTACTOE     *game = &toetactic;
        char                 board9x9[9];
        game->board = board9x9;
        game_init(game);
     /* WARNING: contents of game NOT address of struct */
```

Wopr: additional functions

```
gets(char *string)    # No system calls allowed

puts(char *string)    # No system calls allowed

strcasecmp(char *s1, char *s2)
                      # -1:s1<s2; 0:s1==S2; 1:s1>s2
```

ANSI C: gets and puts

ANSI C Language function: char *gets(char *s) where

char *s is a pointer to a pre-allocated string of bytes.

Gets returns the original pointer *s passed in.

Gets inputs each character and echos it until a newline is encountered (0x0a). The newline is not saved in the final string. The returned string is null terminated.

ANSI C Language function: int puts(char *s) where

char *s is a pointer to a string of bytes to be printed.

Puts prints each character until a null is encountered (0x0a) in the string. A newline is then also printed to the console.

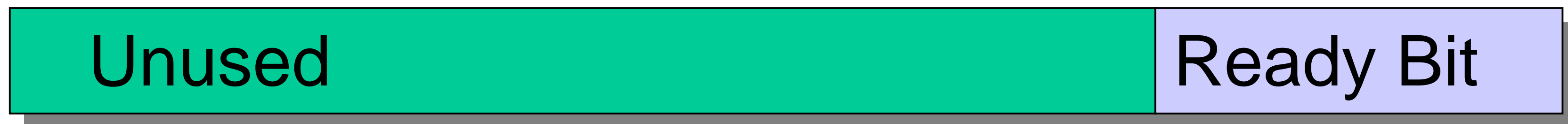
Puts returns the number of characters written to the console.

Rx: Memory Mapped char i/o

(Appendix A-36)

IF Ready bit is true THEN there is a new data character

Receiver control status: memory address 0xffff0000



Receiver data: memory address 0xffff0004

```
Rx: li   $t0,0xffff0000
    lw   $t1,0($t0)          #get rx status
    andi $t1,0x0001          #ready?
    beq  $t1,$zero,Rx        #no
    lbu  $v0,4($t0)          #yes - get byte
```

Tx: Memory Mapped character i/o

IF Tx Ready bit is true THEN ok to output a character

Transmitter control status: memory address 0xffff0008

Transmitter data: memory address 0xffff000c

```
Tx:  li   $t0,0xffff0008
     lw   $t1,0($t0)          #get tx status
     andi $t1,0x0001          #ready?
     beq  $t1,$zero,Tx        #no
     stb  $a0,4($t0)          #yes - put byte
```

Rx_line: Read a line from the console.

```
#Make sure -mapped_io is enabled on spim

rx_line:
     la   $s0, rx_buffer          #string pointer
     li   $t1, 0xffff0000
rx_line1:
     lw   $t2,0($t1)              # ready?
     andi $t2,$t2,1
     beq  $t2,$0,rx_line1         #no - loop

     lbu  $t2,4($t1)              #yes - get char
     sb   $t2,0($s0)              #..store it

     addi $t2,$t2,-10             #carrage return?
     beq  $t2,$0,rx_done          #yes - make it zero
     addi $s0,$s0,1               #next string addr
     j    rx_line1
```

Sun Microsystems SPARC Architecture

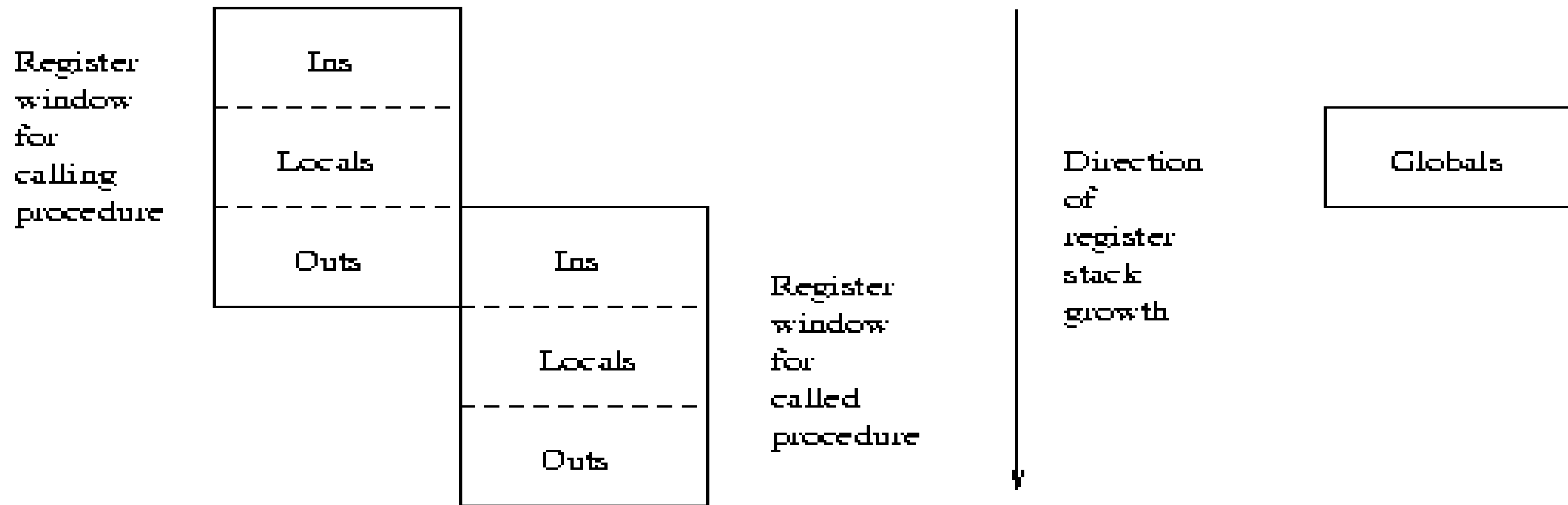
- In 1987, Sun Microsystems introduced a 32-bit RISC architecture called SPARC.
- Sun's UltraSparc workstations use this architecture.
- The general purpose registers are 32 bits, as are memory addresses.
- Thus 2^{32} bytes can be addressed.
- In addition, instructions are all 32 bits long.
- SPARC instructions support a variety of integer data types from single bytes to double words (eight bytes) and a variety of different precision floating-point types.

SPARC Registers

- The SPARC provides access to 32 registers
  - regs 0 %g0 ! global constant 0 (MIPS $zero, $0)
  - regs 1-7 %g1-%g7 ! global registers
  - regs 8-15 %o0-%o7 ! out (MIPS $a0-$a3,$v0-$v1,$ra)
  - regs 16-23 %L0-%L7 ! local (MIPS $s0-$s7)
  - regs 24-31 %i0-%i7 ! in registers (caller’s out regs)

- The global registers refer to the same set of physical registers in all procedures.
- Register 15 (%o7) is used by the call instruction to hold the return address during procedure calls (MIPS ($ra)).
- The other registers are stored in a register stack that provides the ability to manipulate register windows.
- The local registers are only accessible to the current procedure.

SPARC Register windows



- When a procedure is called, parameters are passed in the out registers and the register window is shifted 16 registers further into the register stack.
- This makes the in registers of the called procedure the same as the out registers of the calling procedure.
- in registers: arguments from caller (MIPS %a0-$a3)
- out registers: When the procedure returns the caller can access the returned values in its out registers (MIPS $v0-%v1).

SPARC instructions

Arithmetic

```
add %l1, %i2, %l4       ! local %l4 = %l1 + i2
add %l4, 4, %l4         ! Increment %l4 by four.
mov 5, %l1              ! %l1 = 5
```

Data Transfer

```
ld [%l0], %l1           ! %l1 = Mem[%l0]
ld [%l0+4], %l1         ! %l1 = Mem[%l0+4]
st %l1, [%l0+12]        ! Mem[%l0+12]= %l1
```

Conditional

```
cmp %l1, %l4     ! Compare and set condition codes.
bg  L2           ! Branch to label L2 if %l1 > %l4
nop              ! Do nothing in the delay slot.
```

SPARC functions

Calling functions

```
mov %l1, %o0          ! first parameter = %l1
mov %l2, %o1          ! second parameter = %l2
call fib              ! %o0=.fib(%o0,%o1,…%o7)
nop                   ! delay slot: no op
mov %o0, %l3          ! %i3 = return value
```

Assembler

```
gcc hello.s           ! executable file=a.out
gcc hello.s -o hello  ! executable file=hello
gdb hello             ! GNU debugger
```

SPARC Hello, World.

```
      .data
hmes:.asciz Hello, World\n"
      .text
      .global main    ! visible outside
main:
      add    %r0,1,%%o0      ! %r8 is %o0, first arg
      sethi  %hi(hmes),%o1 ! %r9, (%o1) second arg
      or     %o1, %lo(hmes),%o1
      or     %r0,14,%o2      ! count in third arg
      add    %r0,4,%g1       ! system call number 4
      ta 0                   ! call the kernal

      add    %r0,%r0,%o0
      add    %r0,1,%g1       ! %r1, system call
      ta 0                   ! call the system exit
```

gdb: GNU debugger basics

This is the symbolic debugger for the gcc compiler. So keep all your source files and executables in the same current working directory.

gcc hello.s	Assemble the program hello.s and put the executable in a.out (all files that end in ".s" are assembly files).
gdb a.out	Start the debugger and read the a.out file.
h	gdb Help command: lists all the command groups.
info files	shows the program memory layout (.text, .data, …)
info var	shows global and static variables (_start)
b _start	set the first breakpoint at beginning of program
info break	displays your current breakpoints
r	Start running your program and it will stop at _start

gdb: register & memory contents

Command	Description
info reg	displays the registers
set $L1=0x123	set the register %L1 to 0x123
display $L1	display register %L1 after every single step
info display	show all display numbers
undisplay <number>	stop displaying item <number>
diss 0x120 0x200	dissassemble memory location 0x120 to 0x200
x/b 0x120	display memory location 0x120 as a byte
x/4b 0x120	display memory location 0x120 as four bytes
x/4c 0x120	display memory location 0x120 as four characters
x/s 0x120	display memory location 0x120 as a asciiz string
x/h 0x120	display memory location 0x120 as a halfword
x/w 0x120	display memory location 0x120 as a word

gdb: single stepping

si Single step exactly one instruction

n Single step a single source line but do NOT enter the subroutine.

b *0x2064 This sets a Breakpoint in your program at address 0x2064. Set as many as you need.

info break Display all the breakpoints

c Continue running the program until the next breakpoint. Set more breakpoints or do more “si” or restart program “r”

d Delete all break points.

set args <command_line_args> set the args which are passed to argv & argc

q Quit debugging.

RISC Project: Due last day of lecture

100 points:
Objective: learn structures, pointer, & RISC architecture.

(1) MIPS & C for "reverse TicTacToe" (as explained earlier)

10 points: in class demo before Last Lecture. Limited number of openings. Earlier the better. Must ask beforehand.

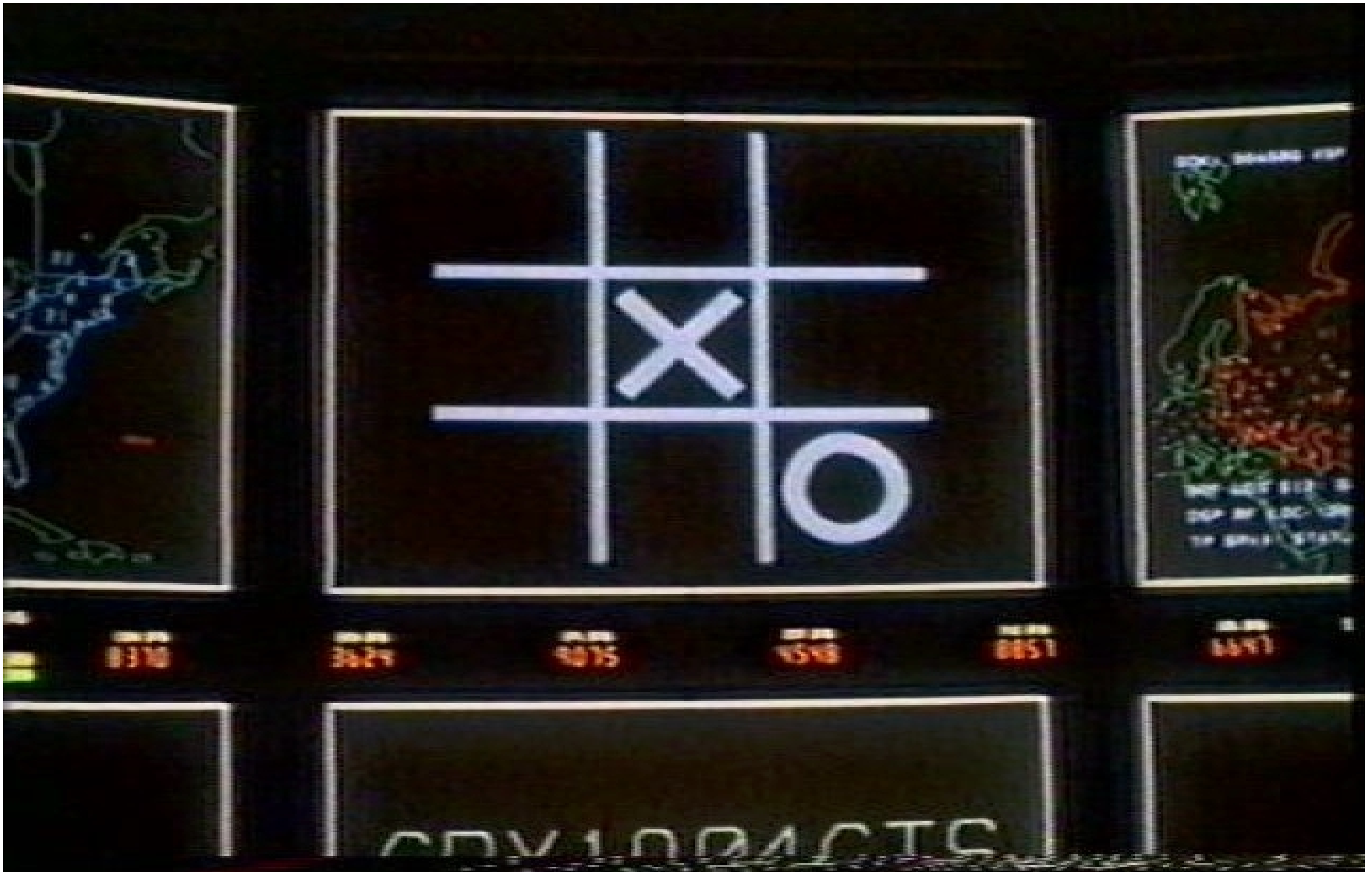
50 points: Objective: learn alternative RISC architecture.

(1) Sun SPARC "reverse TicTacToe"

(2) Can only use kernal calls: "ta 0"

(4) Detailed flowchart of get_move( ) function.

(3) Detailed write up of SPARC instruction binary formats, syntax & semantics, and explain SPARC architecture.



Reverse Tic-Tac-Toe: http://tictactoe.javagamz.com/toetactic.html

Tic-Tac-Toe history: http://home.capecod.net/~pbaum/ttt/intro.htm

Movie References: http://www.imsai.net/Movies/WarGames.htm

http://www-public.rz.uni-duesseldorf.de/~ritterd/wargames/pix.htm

Technical SPARC CPU resources: http://www.users.qwest.net/~eballen1/sparc.tech.links.html

http://www.sunfreeware.com