Functions and Text

David Marchant
Based on slides by Troels Henriksen

2025-09-10

Summary

Hopefully from the material you've learnt:

- How we store data in C and what it represents
- How to read/write data in C

The problem with procedures

```
f(a,b,c);
g(x,y,z);
h(a,y,c);
```

- Calling a procedure (or function) requires jumping to *different code*.
- But:
  - How do we pass arguments to the procedure?
  - How does it return its results?
  - How do we prevent the procedure we call from overwriting the registers we are using?

The basic problem

How do we "suspend" execution of the current procedure, let another procedure take over, and resume execution from the *call site* afterwards?

Main instructions: `jal` and `jalr`

Instruction	Meaning
`jal` x_i, k	x_i = PC + 4; PC += k
`jalr` x_i, k (x_j)	x_i = PC + 4; PC = x_j + k
`jalr` x_i, x_j, k	x_i = PC + 4; PC = x_j + k

- Jump-and-Link jumps to an address and stores the call site address in the "link register" x_i.
- Jump-and-Link-Register takes address from register value.
  - Used to *return* from procedure.

Main data structure: Stacks

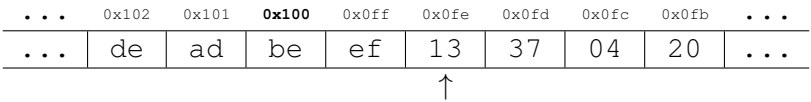
- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.



SP: `0x0fe`

Operation:

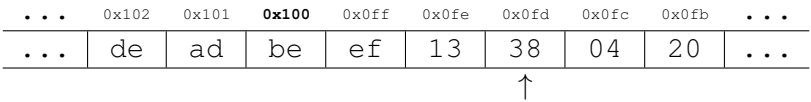
Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.



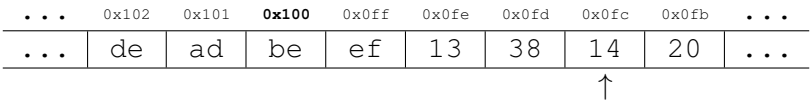
SP: `0x0fd`
Operation: `push(0x38)`
Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.



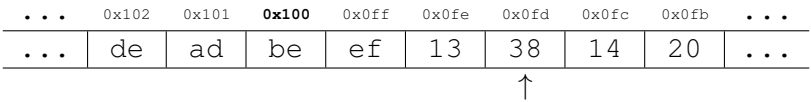
SP: `0x0fc`
Operation: `push(0x14)`
Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.



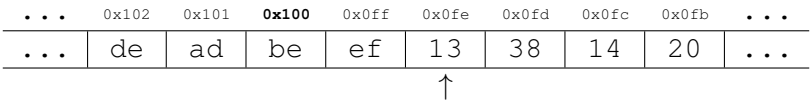
SP: `0x0fd`
Operation: `pop()`
Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.



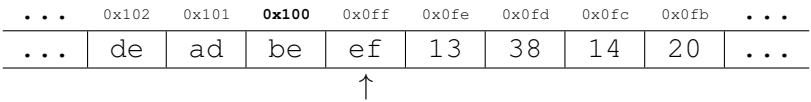
SP: `0x0fe`
Operation: `pop()`
Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

Main data structure: Stacks

- We have seen **structured control flow** encoded using **conditional jumps**.
- Similarly, memory is a flat array of bytes, and it is up to us where we put things.
  - Freedom requires **discipline**.

A stack at machine level

A stack is a region of memory where a *stack pointer* points to the top element.



SP: `0x0ff`
Operation: `pop()`
Stacks can grow from either low to high addresses, or high to low (as above, and in RISC-V).

The call stack

A program can use many stacks for various things, but when we say *the stack*, we almost alway means *the call stack*.

High level idea

- When we **call a function**, we *push* enough information to the stack to allow the called function to resume us when we are done.
- When we **return from a function**, we *pop* information from the stack to resume execution of the caller.
- Call stack is essentially a **queue of functions waiting to resume execution.**

The call stack

A program can use many stacks for various things, but when we say *the stack*, we almost alway means *the call stack*.

High level idea

- When we **call a function**, we *push* enough information to the stack to allow the called function to resume us when we are done.
- When we **return from a function**, we *pop* information from the stack to resume execution of the caller.
- Call stack is essentially a **queue of functions waiting to resume execution.**

Important

- The call stack is **not special hardware**, but uses ordinary memory.
- By RISC-V **convention**, `sp` register points to top of stack (the *stack pointer*).
- Grows from high address to low address.
  - To **pop**: *increment* `sp`.
  - To **push**: *decrement* `sp`.

Example for leaf procedure that calls no other procedures

```
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
```

Example for leaf procedure that calls no other procedures

```
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -12   # make room for 3 words
sw t0, 8(sp)       # save old value of t0
sw t1, 4(sp)       # save old value of t1
sw s4, 0(sp)       # save old value of s4
```

Example for leaf procedure that calls no other procedures

```
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -12   # make room for 3 words
sw t0, 8(sp)       # save old value of t0
sw t1, 4(sp)       # save old value of t1
sw s4, 0(sp)       # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
```

Example for leaf procedure that calls no other procedures

```
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -12   # make room for 3 words
sw t0, 8(sp)       # save old value of t0
sw t1, 4(sp)       # save old value of t1
sw s4, 0(sp)       # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
addi a0, s4, 0     # return value in a0
```

Example for leaf procedure that calls no other procedures

```
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -12   # make room for 3 words
sw t0, 8(sp)       # save old value of t0
sw t1, 4(sp)       # save old value of t1
sw s4, 0(sp)       # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
addi a0, s4, 0     # return value in a0
lw s4, 0(sp)       # restore s4
lw t1, 4(sp)       # restore t1
lw t0, 8(sp)       # restore t0
addi sp, sp, 12    # pop 3 words from stack
```

Example for leaf procedure that calls no other procedures

```
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -12   # make room for 3 words
sw t0, 8(sp)       # save old value of t0
sw t1, 4(sp)       # save old value of t1
sw s4, 0(sp)       # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
addi a0, s4, 0     # return value in a0
lw s4, 0(sp)       # restore s4
lw t1, 4(sp)       # restore t1
lw t0, 8(sp)       # restore t0
addi sp, sp, 12    # pop 3 words from stack
jalr zero, 0(ra)   # jump to call site
```

Motivation for calling conventions

Terminology

Caller The procedure making the procedure call (jumping *from*).

Callee The procedure being called (jumping *to*).

Motivation for calling conventions

Terminology

Caller The procedure making the procedure call (jumping *from*).

Callee The procedure being called (jumping *to*).

- Large modular programs consist of procedures that
  - can be called from *any other procedure*.
  - can call *any other procedure* without knowing how they work internally.

Calling convention

A set of rules for how to pass arguments to another procedure, what that procedure can expect of its environment, and how the callee should clean up and return to the caller.

Calling convention

A set of rules for how to pass arguments to another procedure, what that procedure can expect of its environment, and how the callee should clean up and return to the caller.

- **Differs between architectures and operating systems.**
  - We'll ignore lots of features that are important in practice, e.g. exception handlers, destructors, or debugging information.

Calling convention

A set of rules for how to pass arguments to another procedure, what that procedure can expect of its environment, and how the callee should clean up and return to the caller.

- **Differs between architectures and operating systems.**
  - We'll ignore lots of features that are important in practice, e.g. exception handlers, destructors, or debugging information.
- **Basics:**
  - Caller puts arguments in `a0`–`a7`.
  - Caller puts return address in `ra`.
  - Callee puts return value in `a0`.
- But what about the other registers?

Caller-save and callee-save registers (simplified)

Caller-save registers, e.g. `t0–t6`

From callers perspective

- *Not* preserved by procedure calls.
- Must be saved on stack before call if we want to preserve value.

From callees perspective

- May be overwritten freely.
- Can have any value when returning.

Caller-save and callee-save registers (simplified)

Caller-save registers, e.g. `t0–t6`

From callers perspective

- *Not* preserved by procedure calls.
- Must be saved on stack before call if we want to preserve value.

From callees perspective

- May be overwritten freely.
- Can have any value when returning.

Callee-save registers, e.g. `s0–s11`

From callers perspective

- Preserved by procedure calls.

From callees perspective

- Must save value on stack before overwriting.
- Must restore original value before returning.

Full table of integer registers

Register	Name	Description	Saver
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5–7	t0–t2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–a1	Arguments/return values	Caller
x12–17	a2–a7	Arguments	Caller
x18–27	s2–s11	Saved registers	Callee
x28–31	t3–t6	Temporaries	Caller

- 16 are caller-save.
- 13 are callee-save.

Leaf procedure, saving only callee-saves registers

```
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -4     # make room for 1 word
sw s4, 0(sp)        # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
addi a0, s4, 0      # return value in a0
lw s4, 0(sp)        # restore s4
addi sp, sp, 4      # pop 1 word from stack
jalr zero, 0(ra)    # jump to call site
```

Leaf procedure, saving only callee-saves registers

```
int leaf(int a0,
         int a1,
         int a2,
         int a3) {
  int t0 = a0 + a1;
  int t1 = a2 + a3;
  int s4 = t0 - t1;
  return s4;
}
```

```
leaf:
addi sp, sp, -4     # make room for 1 word
sw s4, 0(sp)        # save old value of s4
add t0, a0, a1
add t1, a2, a3
add s4, t0, t1
addi a0, s4, 0      # return value in a0
lw s4, 0(sp)        # restore s4
addi sp, sp, 4      # pop 1 word from stack
jalr zero, 0(ra)    # jump to call site
```

Can we do even better?

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

Calling the leaf procedure

```
addi a0, t0, 0    # first argument
addi a1, t1, 0    # second argument
addi a2, t2, 0    # third argument
addi a3, t3, 0    # fourth argument
```

```
leaf(t0,t1,t2,t3);
```

Calling the leaf procedure

```
addi a0, t0, 0     # first argument
addi a1, t1, 0     # second argument
addi a2, t2, 0     # third argument
addi a3, t3, 0     # fourth argument
addi sp, sp, -64   # make room for 16 words
```

```
leaf(t0,t1,t2,t3);
```

Calling the leaf procedure

```
addi a0, t0, 0     # first argument
addi a1, t1, 0     # second argument
addi a2, t2, 0     # third argument
addi a3, t3, 0     # fourth argument
addi sp, sp, -64   # make room for 16 words
sw ra, 0(sp)       # save ra
```

```
leaf(t0,t1,t2,t3);
```

Calling the leaf procedure

```
                        addi a0, t0, 0     # first argument
                        addi a1, t1, 0     # second argument
                        addi a2, t2, 0     # third argument
                        addi a3, t3, 0     # fourth argument
                        addi sp, sp, -64   # make room for 16 words
                        sw ra, 0(sp)       # save ra
                        sw t0, 4(sp)       # save t0
leaf(t0,t1,t2,t3);
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
lw ra, 0(sp)        # restore ra
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
lw ra, 0(sp)        # restore ra
lw t0, 4(sp)        # restore t0
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
lw ra, 0(sp)        # restore ra
lw t0, 4(sp)        # restore t0
...
lw a7, 60(sp)       # restore a7
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
lw ra, 0(sp)        # restore ra
lw t0, 4(sp)        # restore t0
...
lw a7, 60(sp)       # restore a7
addi sp, sp, 64     # restore stack pointer
...                 # continue on
```

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0      # first argument
addi a1, t1, 0      # second argument
addi a2, t2, 0      # third argument
addi a3, t3, 0      # fourth argument
addi sp, sp, -64    # make room for 16 words
sw ra, 0(sp)        # save ra
sw t0, 4(sp)        # save t0
...
sw a7, 60(sp)       # save a7
jal ra, leaf        # jump to procedure
lw ra, 0(sp)        # restore ra
lw t0, 4(sp)        # restore t0
...
lw a7, 60(sp)       # restore a7
addi sp, sp, 64     # restore stack pointer
...                 # continue on
```

Should we restore all registers?

Calling the leaf procedure

```
leaf(t0,t1,t2,t3);
```

```
addi a0, t0, 0     # first argument
addi a1, t1, 0     # second argument
addi a2, t2, 0     # third argument
addi a3, t3, 0     # fourth argument
addi sp, sp, -64   # make room for 16 words
sw ra, 0(sp)       # save ra
sw t0, 4(sp)       # save t0
...
sw a7, 60(sp)      # save a7
jal ra, leaf       # jump to procedure
lw ra, 0(sp)       # restore ra
lw t0, 4(sp)       # restore t0
...
lw a7, 60(sp)      # restore a7
addi sp, sp, 64    # restore stack pointer
...                # continue on
```

Should we restore all registers? *Possibly not* `a0` (would overwrite return value)

```
int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

```
mul:
        mv a2, a0
        mv a0, zero
mul_loop:
        beq a2, zero, mul_end
        add a0, a0, a1
        addi a2, a2, -1
        jal zero, mul_loop
mul_end:
        jalr zero, ra, 0
```

```
fact:
        addi sp, sp, -8
        sw ra, 4(sp)
        sw a0, 0(sp)
        beq a0, zero, fact_base
        addi a0, a0, -1
        jal ra, fact
        lw a1, 0(sp)
        jal ra, mul
        jal zero, fact_return
fact_base:
        li a0, 1
fact_return:
        lw ra, 4(sp)
        addi sp, sp, 8
        jalr zero, ra, 0
```

Concept Questions

- One of my goals for the course this year is to see a better conceptual understanding of computing in student responses
- This might not be a style of answer you are used to giving

Why

- In short, because understanding a topic is far more useful than just being able to replicate it.
- LLMs are making raw programming ability less worthwhile
- Future knowledge will be built upon the *concepts*, not on the functions of CompSys

Examples

- Within the subject of networking, what is meant by the concept of a protocol? Describe what the purpose of protocols are, and what they enable us to do within a network. How does this relate to the concept of layering?
- Congratulations, you have started a new job at a small telecommunications company. The company founder has created a new IP protocol that uses trinary (each bit can have 3 values as opposed to binary's 2 values per bit). The founder wishes to see their new trinary based IP protocol replace all currently deployed IPv4 and IPv6 protocols.
  Your job at this new company is to produce a report on how feasible it is for this to be done. What problems would you write about in such a report? As this is not a business exam you do not need to consider things such as financial cost, manpower etc. in any detail, but should instead focus on any technical or design problems.

Argument

- A good answer to these is going to be more discussion based than just being a list of facts
- You should be able to bring in and apply your knowledge to justify your answer, and potentially apply it in unfamiliar ways
- A good structure is to set out assumptions (if any), knowledge, and a conclusion

Assumptions

- Anything not stated, but you think it is sensible to assume
- Could be physical characteristics of the system, or ways in which it is going to be used
- You don't need to state everything, just things that will affect your conclusion
- This is usually short, sometimes not needed at all

Argument

- Identify and explain the relevant concepts
- Extrapolate how they could be applied to any new use cases
- You don't always need to be definite, can be talk in terms of probabilities
- This is the real meat of the answer, put forward your argument

Conclusion

- Good science always has a conclusion
- This should be supported by your discussion and assumptions
- Premises $\rightarrow$ Conclusion

Let's Try A Question

- Why is it commonly accepted practice to follow a coding standard?

SOLO Taxonomy

- Structure of Observed Learning Outcome
- A methodology for assessing student submissions
- Not the be-all and end-all, but it is the closest I've found to how I mark, and is used in design marking guides for the TA's to mark your work.
- Rates answers in terms of:
  - Capacity: How much knowledge has been demonstrated
  - Relating Operation: Is your answer a repetition of the question, or does it a logical induction?
  - Consistency: Does your answer accommodate several pieces of knowledge without contradiction
  - Closure: Does your response answer all problems posed

SOLO Taxonomy

Level	Capacity	Relating Operation	Consistency and Closure
Extended Abstract	Hypotheses	Deduction	No broad inconsistencies
Relational	Interrelations	Induction	No narrow inconsistencies
Multi-structural	Relevant Data	Limited generalisation	Closes too early
Uni-structural	One Data Point	One generalisation	Jumps to conclusions
Pre-structural	Que	Tautology	No consistency

An Example

- Congratulations, you have started a new job at a small telecommunications company. The company founder has created a new IP protocol that uses trinary (each bit can have 3 values as opposed to binary's 2 values per bit). The founder wishes to see their new trinary based IP protocol replace all currently deployed IPv4 and IPv6 protocols.
  Your job at this new company is to produce a report on how feasible it is for this to be done. What problems would you write about in such a report? As this is not a business exam you do not need to consider things such as financial cost, manpower etc. in any detail, but should instead focus on any technical or design problems.

Possible Answer

- Let's assume the creator has also got a device capable of running trinary. This now means he needs to deploy these devices everywhere as current binary systems will not be able to support it.
- Alternatively they could have created an emulator, but this would add massive overhead as everything would need to converted from the binary machine code to whatever system the trinary is written in.
- No existing router would support this as they have IP soldered into them at a hardware level.
- It is not clear what is wrong with IP, as they are sufficient for meeting the modern and expected needs of the internet. Current problems such as fairness or filling the address space do not require a complete rework of the entire system.
- The strength of IP is that everyone uses it. There's no way to address a trinary system in IP unless trinary just uses the same address space, somewhat defeating the point of a new system.
- This is unfeasible as it would in all likelihood be slower to run, difficult to deploy and would fracture the internet for little apparent gain.

Let's Try A Question ... Again

- Review your answer to the previous question
- How well does it meet the SOLO taxonomy?
- Can you identify premises and do they support your conclusion?

Let's Try Some More Questions

- If everything runs in machine code anyway, why do we program in higher level languages like C?

Let's Try Some More Questions

- If higher level languages like C are so good, why don't we invent a CPU that runs uses that as its instruction set instead?

Let's Try Some More Questions

- Is computer science really a science?