

Introduction to Program Synthesis (WS 2024/25)

Chapter 2.3 - Foundations (Program Representation: Stack)

Dr. rer. nat. Roman Kalkreuth

Chair for AI Methodology (**AIM**), Department of Computer Science,
RWTH Aachen University, Germany



Center for
Artificial Intelligence

RWTHAACHEN
UNIVERSITY

Computer Programs: Representations

Stack-based representation

- ▶ Computer program → Sequence of instructions
 - ~> Commonly divided into sub-sequences in concurrent fashion
 - ~> Execution can be organized with a stack
- ▶ Stack → Linear data structure to represent sequential execution order
 - ~> Data insertion policy → Last In First Out (LIFO)
 - ~> By default has three operations → `push()`, `pop()`, `top()`
 - ~> Can be also extended with nesting
- ▶ **Stack-based programming** → Most operations to handle variables are accomplished with one or more stacks
 - ~> Provision of additional operators in addition to the above-mentioned ones

Computer Programs: Representations

Execution of programs

1. Sequence of instructions → stored in **memory**
2. Address in memory of the first location is copied to the **instruction pointer (IP)**
3. **CPU** → sends the address of the instruction to the **address bus**
4. **CPU** → sends a read signal to the control bus
5. **Memory** → sends a copy of the bits stored at the current address which are stored on the **data bus**
 ~> Instruction is then loaded into the **instruction register**
6. **Instruction pointer** → automatically incremented to the address that stores the next instructions
7. Instruction in the instruction register is executed
8. Go back to step 3

Steps 3, 4, 5 → **instruction fetch**

Steps 3-8 → **execution cycle**

Computer Programs: Representations

Execution of programs

```
1  main :
2      mov     eax,1
3      mov     ebx,0
4      mov     edx,1
5      mov     ecx,6          ; counter
6
7  L1:
8      mov     eax,ebx        ;  $eax = ebx + edx$ 
9      add     eax,edx
10     mov     ebx,edx
11     mov     edx,eax
12     dec     ecx
13     jnz     L1
14     exit
```

Listing: Assembly program to calculate the first seven numbers of the Fibonacci number sequence (1,1,2,3,5,8,13)

Computer Programs: Representations

CPU instructions

- ▶ Instructions → Atomic elements of computer programs
 - ~> Categories → **Computational**, **Load/Store**, **Jump** and **Branch**, **Floating Point**
- ▶ **Stack register** → Stores the memory address to the **call stack** of the currently executed program
 - ~> Also known as **stack pointer** (SP), **program counter** (PC) or **instruction pointer** (IP)
 - ~> Always points to the top of the stack
- ▶ **Stack frame** → Representation of a function call and the corresponding argument data
 - ~> Frame of data that is pushed onto the stack

Computer Programs: Representations

CPU organization

- ▶ Programs can be stored either in computer memory or CPU registers
- ▶ **Stack organization** → Registers of the CPU are organized and structured with a stack
 - ~> Register unit or the memory of the CPU is organized with a stack
- ▶ Two types of stacks are commonly used by the CPU
 - ~> **Register stack**
 - ~> **Memory stack**

Computer Programs: Representations

CPU instructions

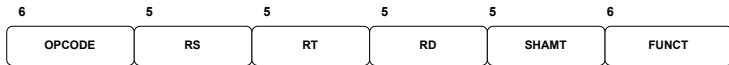


Figure: MIPS R-Type 32 bit instruction encoding

- ▶ opcode (6 bits) → specifies the operation to be executed
- ▶ r_s (5 bits) → register address of the first operand
- ▶ r_t (5 bits) → register address of the second operand
- ▶ r_d (5 bits) → register address of the destination of the result
- ▶ shamt (5 bits) → number of shifts (only for shift instructions)
- ▶ funct (6 bits) → function code for augmentation of the opcode

Computer Programs: Representations

CPU instructions

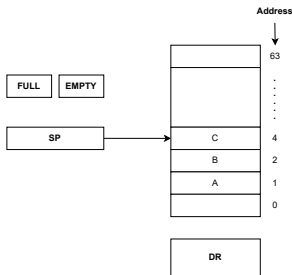


Figure: 64 word register stack

- ▶ CPU registers are organized with a stack
- ▶ Stack pointer \rightarrow 6 bit register, because $2^6 = 64$
- ▶ Data Register (DR) \rightarrow Data is popped from or pushed into the stack from here
 - \rightsquigarrow push \rightarrow Increment stack pointer
 - \rightsquigarrow pop \rightarrow Decrement stack pointer

Computer Programs: Representations

CPU instructions

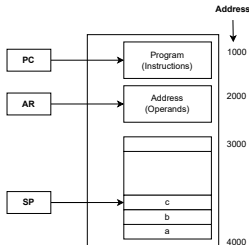


Figure: Memory stack illustration

- ▶ Primary memory (RAM) is organized with a stack for program execution:
 - ~ Program Counter (PC) → points to the address of the next instruction
 - ~ Address Register (AR) → points to an element within the memory stack (used to read operands)
 - ~ Stack Pointer (SP) → points to the top of the stack

Computer Programs: Representations

CPU instructions

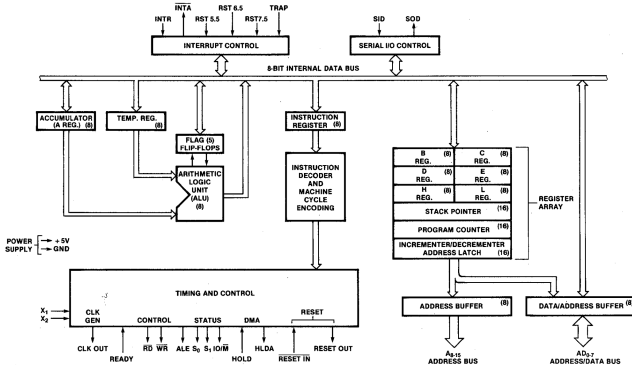


Figure: Intel 8085 microprozessor block diagram

► Specifications:

- 6.500 transistors - 8 Bit data bus - 16 bit address (64 KB directly addressable)
- 5,0 MHz clock frequency → 400.000 instructions per sec

Computer Programs: Representations

Call Stack

- ▶ Functions of a program → Can be considered sub-programs calls
 - ~> Instructions sequentially fetched from memory
- ▶ **Call stack (CS)** → Also called program, execution or procedure stack
 - ~> Holds all function calls of a program
 - ~> Keeps track of these sub-calls within program execution
 - ~> Multiple stacks can be used for multi-threading
- ▶ High-level call stack vs processor's call stack
 - ~> **Low-level stack** → works with addresses and values at the byte/word level
 - ~> **High-level stack** → Stores and keeps track of function calls of a high-level language

Computer Programs: Representations

CPU instructions

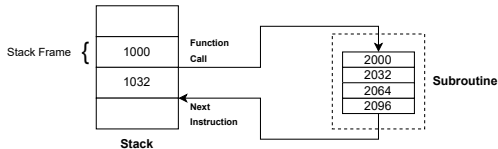


Figure: Call stack illustration

- ▶ Function call → new stack frame is created
 - ↪ Stack frame contains function's data
 - ↪ That is parameters, locals and return address
- ▶ Function call procedure in C:
 1. Stack Frame is pushed into the stack
 2. Instructions (of subroutine) are executed
 3. Stack Frame is popped from the stack
 4. Return address is assigned to the program counter
- ▶ Return address is pushed onto the stack and retrieved after the call
 - ↪ Program counter is modified when the function is called

Computer Programs: Representations

Stack-based programming

- ▶ Use of stack-based or stack-oriented programming languages
 - ~> Central paradigm → Stack is fundamental for the programming model
- ▶ Stack(s) for passing argument and return values
- ▶ C/C++/Java → memory can **either** be allocated on the **stack** or the **heap**

Computer Programs: Representations

Forth

- ▶ **Forth** → Stack-based programming language¹
- ▶ Uses postfix notation → (op1 op2 func) → (1 4 +)
- ▶ Provides several types of stacks
 - ~ Data stack → stores characters, cells, addresses, and double cells
 - ~ Floating point stack → stores floating point numbers
 - ~ Return stack → stores return addresses
 - ~ Local stack → stores local variables

Computer Programs: Representations

Forth

- ▶ Cyclic permutation and variation of stack elements
- ▶ Extended stack manipulation operation set:

~> dup(a -- a a)

~> rot(a --)

~> swap(a b -- b a)

~> roll(a b c -- b c a)

~> over (a b -- a b a)

~> nip (a b -- b) → swap drop

~> tuck (a b -- b a b) → swap over

Computer Programs: Representations

Forth

```
1 def fibonacci_iter(n):  
2     f = [0,1,1]  
3     for i in range(2, n):  
4         f.append(f[-1] + f[-2])  
5     return f[-1]
```

```
1 : fibonacci_iter ( n1 — n2 )  
2   1 0 rot 0 ?do dup over + loop drop  
3 ;
```

- ▶ **0 1 rot (0 1 n) [n,0,1]**
 ~ Initial setting of the sequence & loop count is rotated to the top of the stack
- ▶ **0 ?do ... loop (n --) [0,1]**
 ~ Count is taken from the stack, two item are left for the loop body
- ▶ **dup (a -- a a) [0,1,0]**
 ~ Item under the top of the stack is copied to the top
- ▶ **over (a b -- a b a) [0,1,0,1]**
 ~ Item under the top of the stack is copied to the top
- ▶ **+ (a b -- a+b) [0,1,1]**
 ~ The two items are added and the sum is pushed into the stack
- ▶ **drop (a --)**
 ~ Removes the extra term before returning the result

Computer Programs: Representations

Forth

```
1 def fibonacci_rec(n):
2     if n < 2:
3         return n
4     else:
5         f = fibonacci_rec(n-1) + fibonacci_rec(n-2)
6     return f
```

```
1 : fib-rec ( n — f )
2   dup 2 u< if exit then
3   1- dup recurse swap 1- recurse +
4   ;
```

- ▶ `dup 2 u< if exit then`
 - ↪ `n` is returned in case that it is 0 or 1
 - ↪ `u<` is an unsigned comparison
- ▶ `1- (n - n-1)`
 - ↪ Decrement operator
- ▶ `recurse`
 - ↪ Performs a recursive call of the function