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

RWTH AACHEN UNIVERSITY

Computer Programs: Representations

Stack-based representation

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

Computer Programs: Representations

Execution of programs

1. Sequence of instructions $\rightarrow$ stored in **memory**
2. Address in memory of the first location is copied to the **instruction pointer (IP)**
3. **CPU** $\rightarrow$ sends the address of the instruction to the **address bus**
4. **CPU** $\rightarrow$ sends a read signal to the control bus
5. **Memory** $\rightarrow$ sends a copy of the bits stored at the current address which are stored on the **data bus**
   - $\rightsquigarrow$ Instruction is then loaded into the **instruction register**
6. **Instruction pointer** $\rightarrow$ 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 $\rightarrow$ **instruction fetch**
Steps 3-8 $\rightarrow$ **execution cycle**

Computer Programs: Representations

Execution of programs

```
main:
    mov    eax,1
    mov    ebx,0
    mov    edx,1
    mov    ecx,6          ; counter
l1:
    mov   eax,ebx         ; eax = ebx + edx
    add   eax,edx
    mov   ebx,edx
    mov   edx,eax
    dec ecx
    jnz L1

    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** $\rightarrow$ Registers of the CPU are organized and structured with a stack
  - $\rightsquigarrow$ Register unit or the memory of the CPU is organized with a stack
- Two types of stacks are commonly used by the CPU
  - $\rightsquigarrow$ **Register stack**
  - $\rightsquigarrow$ **Memory stack**

Computer Programs: Representations
CPU instructions

6	5	5	5	5	6
OPCODE	RS	RT	RD	SHAMT	FUNCT

Figure: MIPS R-Type 32 bit instruction encoding

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

Computer Programs: Representations

CPU instructions

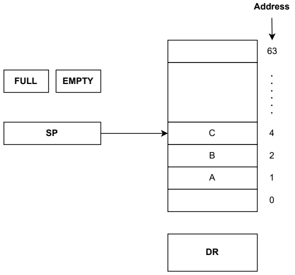

Figure: 64 word register stack

- 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

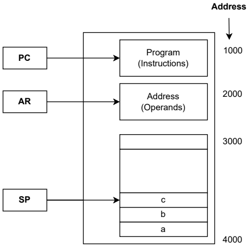

Figure: Memory stack illustration

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

Computer Programs: Representations

CPU instructions

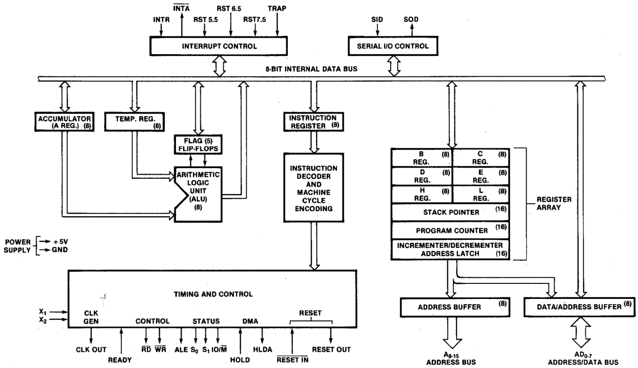

Figure: Intel 8085 microprozessor block diagram

- 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

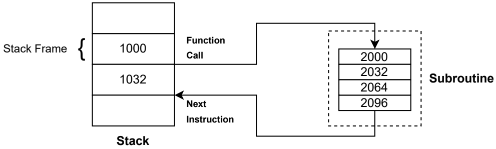

Figure: Call stack illustration

- Function call $\rightarrow$ new stack frame is created
  - $\leadsto$ Stack frame contains function's data
  - $\leadsto$ 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
  - $\leadsto$ Program counter is modified when the function is called

Computer Programs: Representations

Stack-based programming

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

Computer Programs: Representations
Forth

- **Forth** $\rightarrow$ Stack-based programming language[1]
- Uses postfix notation $\rightarrow$ `(op1 op2 func)` $\rightarrow$ $(1\ 4\ +)$
- Provides several types of stacks
  - $\rightsquigarrow$ Data stack $\rightarrow$ stores characters, cells, addresses, and double cells
  - $\rightsquigarrow$ Floating point stack $\rightarrow$ stores floating point numbers
  - $\rightsquigarrow$ Return stack $\rightarrow$ stores return addresses
  - $\rightsquigarrow$ Local stack $\rightarrow$ stores local variables

[1] https://gforth.org/

Computer Programs: Representations
Forth

- Cyclic permutation and variation of stack elements
- Extended stack manipulation operation set:
  - $\leadsto$ `dup(a -- a a)`
  - $\leadsto$ `rot(a --)`
  - $\leadsto$ `swap(a b -- b a)`
  - $\leadsto$ `roll(a b c -- b c a)`
  - $\leadsto$ `over ( a b -- a b a )`
  - $\leadsto$ `nip ( a b -- b )` $\rightarrow$ `swap drop`
  - $\leadsto$ `tuck ( a b -- b a b )` $\rightarrow$ `swap over`

Computer Programs: Representations

Forth

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

```forth
: fibonacci_iter ( n1 -- n2 )
  1 0 rot 0 ?do dup over + loop drop
;
```

- `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

```python
def fibonacci_rec(n):
    if n < 2:
        return n
    else:
        f = fibonacci_rec(n-1) + fibonacci_rec(n-2)
    return f
```

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

- 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

Computer Programs: Representations
Push

- **Stack-based language** $\rightarrow$ Do not need grammars nor parsing to construct a program
- Simple program modification and high degree of flexibility

Computer Programs: Representations

Push

- **Push**[SR02] → stack-based programming language designed for artificial **auto-constructive evolution**
- Extension of traditional stack-based languages such as Forth
- Meets the requirements for auto-constructive evolution of programs:
  - **Expressiveness** → Programs should be easily representable with multiple data types, modules and complex control structures
  - **Self-manipulating** → Programs should be able to manipulate and produce other programs
  - **Syntactically uniform** → Facilitating the development of self-manipulating code and simplifying variation for search heuristics

Computer Programs: Representations
Push

- **Autoconstructive Evolution** $\rightarrow$ Process of autonomous evolution
  - Responsible for the aspects of the evolutionary process itself
- **Autoconstructive evolution system** $\rightarrow$ Evolutionary computation based system that constructs its own mechanism for reproduction and mutation

Computer Programs: Representations

Push

- Design goals $\rightarrow$ maximize **semantic flexibility** and **minimize fragility** for syntax errors
- Push uses various stacks for different purposes:
  - Code stack
  - Boolean stack
  - Float stack
  - Integer stack

Computer Programs: Representations
Forth

```
(CODE QUOTE (INTEGER 2 3 +) DO)
```

Listing: Code "encapsulation" in Push

```
(code quote
(quote (pop 1)
quote (code dup integer dup 1 - do *)
integer dup 2 < if)
do)
```

Listing: Recursive calculation of the factorial

- Encapsuling of code which is put onto the **code stack**
- Enables **recursion** and **modularity** for variation

References

[SR02] Lee Spector and Alan J. Robinson. "Genetic Programming and Autoconstructive Evolution with the Push Programming Language". In: *Genet. Program. Evolvable Mach.* 3.1 (2002), pp. 7–40. DOI: 10.1023/A:1014538503543. URL: https://doi.org/10.1023/A:1014538503543.