Transcript

Introduction to Operating Systems VO

Version v1.0

Daniel Kocher

Daniel.Kocher@stud.sbg.ac.at

Salzburg, October 8, 2014

Myth Busting

Myth: I prefer programming language X because it makes it more convinient to implement my application.

Answer: Wrong!

The fundamental problem of programming is to establish functional correctness and adequate performance. Different languages provide different tools of automating the process of establishing correctness and performance. The language should be chosen based on that insight.

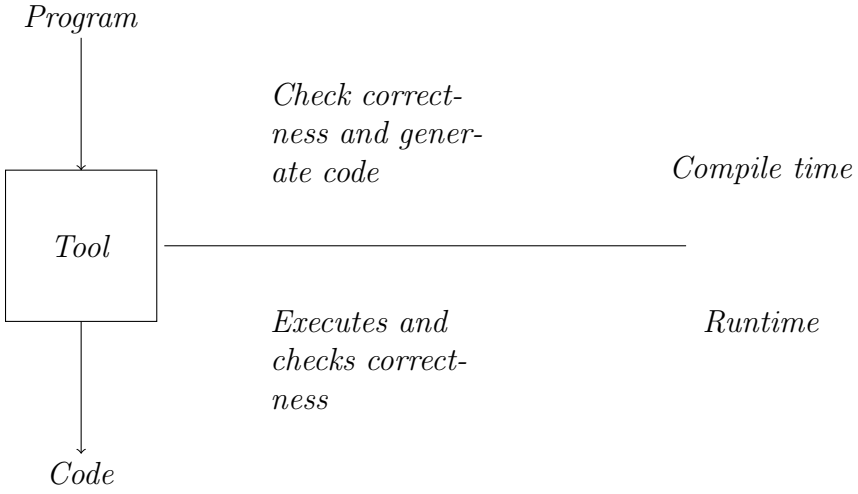

Figure 1: Process of establishing correctness.

Ultimate goal:
A Compiler checking correctness in such a sense that no exception is thrown while executing in any case. But this is infeasible (mathematical proof possible).

Hardware Exception: Interrupts, a mechanism to stop memory accesses in hardware.

Myth: I like garbage-collected languages like Java because they free me from memory management.

Answer: Wrong!

A garbage collector (GC) provides safe deallocation of unneeded memory but the programmer still needs to say what is unneeded, otherwise the system will run out of memory (memory leak).

General runtime complexity of a garbage collector: the size of the heap.

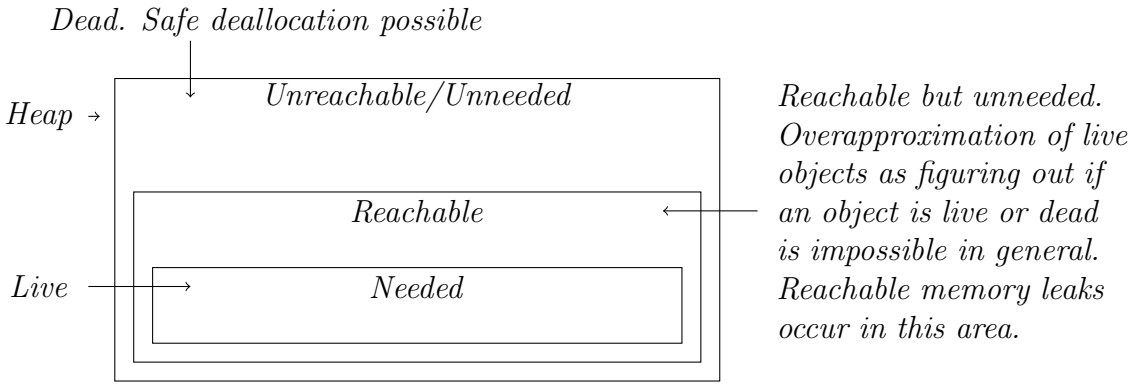

Figure 2: Unreachable, reachable and needed set of a program.

Multicore

Amdahl's Law: P represents program parallelism on N cores

- $S(N) = N$ if $P = 100\%$. This is ideal multicore scalability.
- In general: $S(N) = \frac{1}{(1-P)\cdot\frac{P}{N}}$

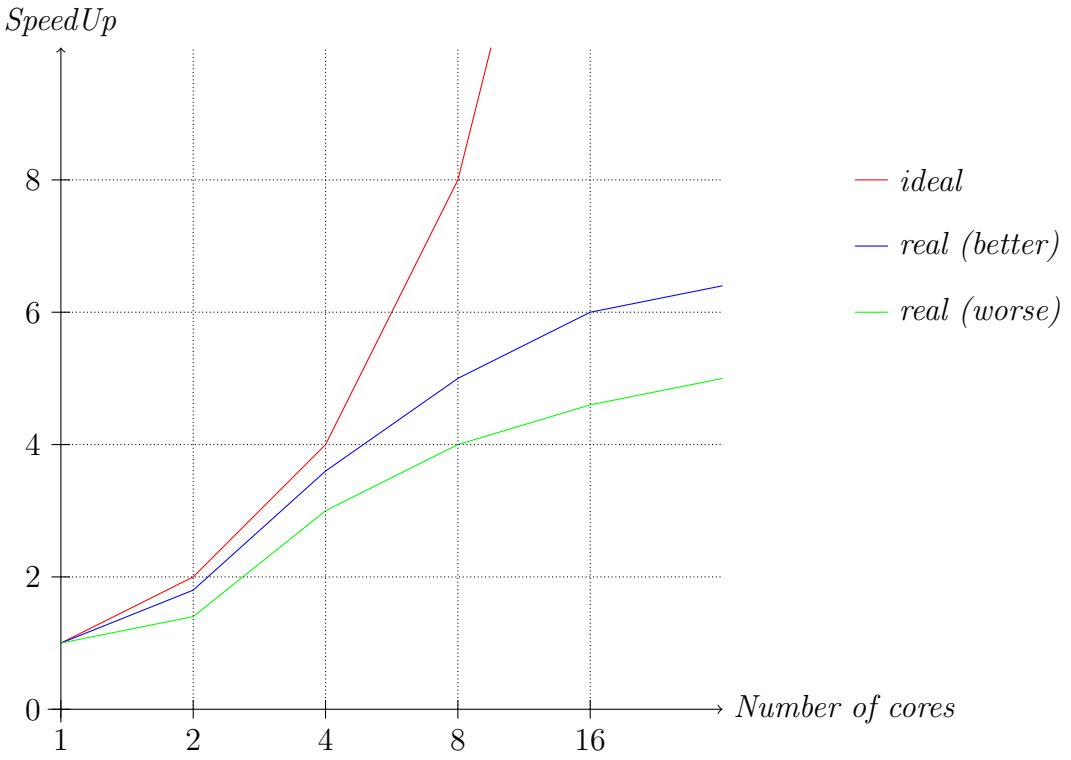

Figure 3: Amdahl's law plot.

Sequential vs. Parallelized Code

The bottleneck of parallelized is the memory bus. It is limiting execution even if a problem can be perfectly parallelized without any side effects. Thus a parallelization factor of 100% is infeasible. Any shared resource at any level of an architecture creates a limitation.

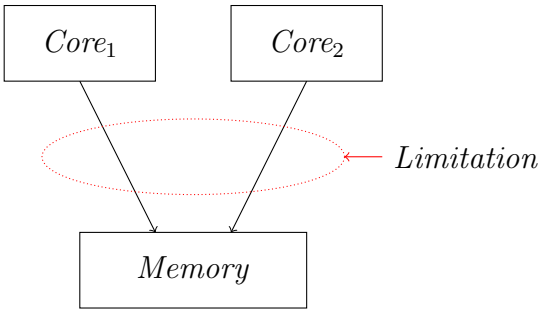

Figure 4: Bottleneck of parallelized code.

Architecture

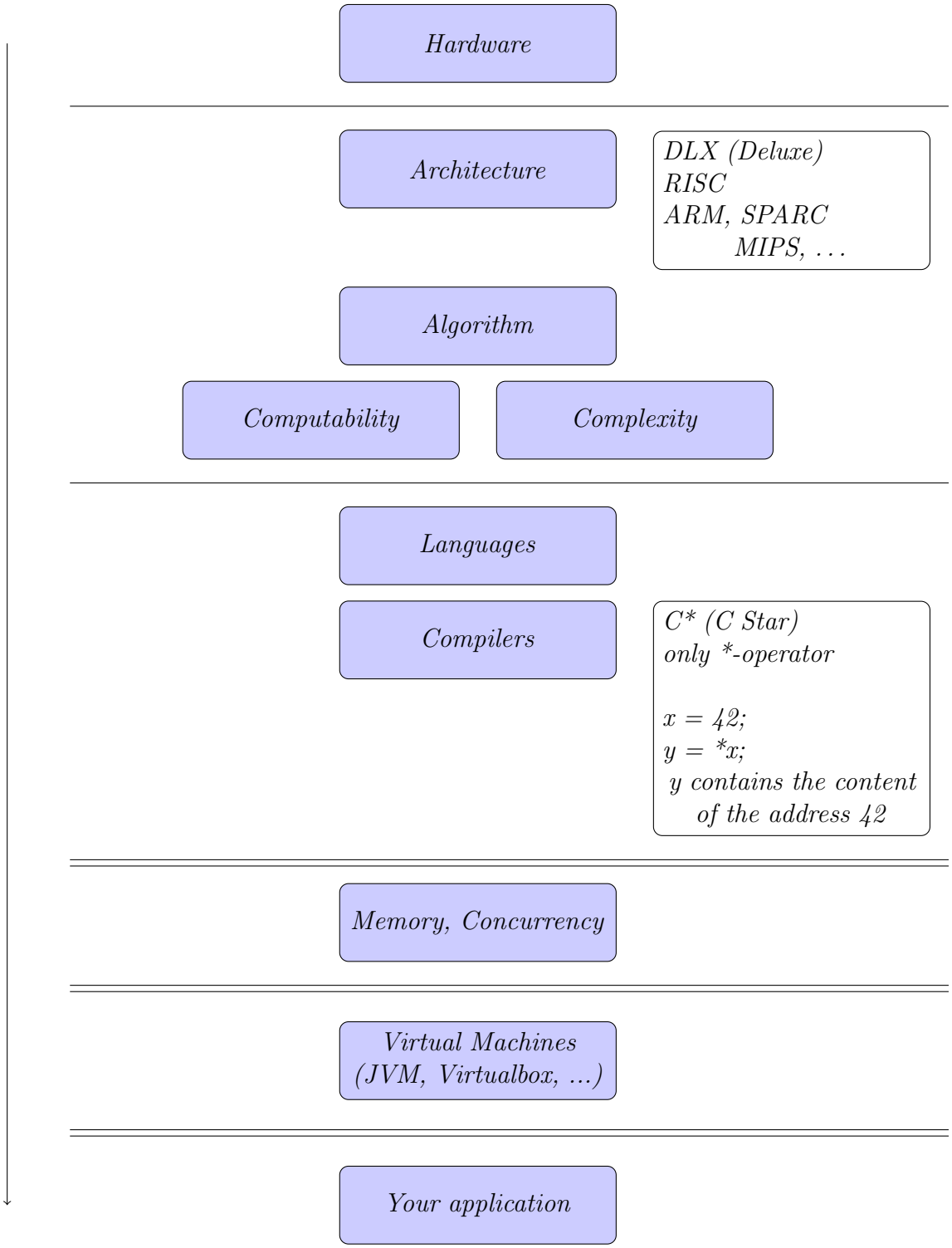

Figure 5: Architecture hierarchy (top-down.

Von Neumann Architecture

Introduced in 1945. This is a stored program computer: data = program. The Fetch-Decode-Execute cycle (see Figure 7) modifies the state of the machine.

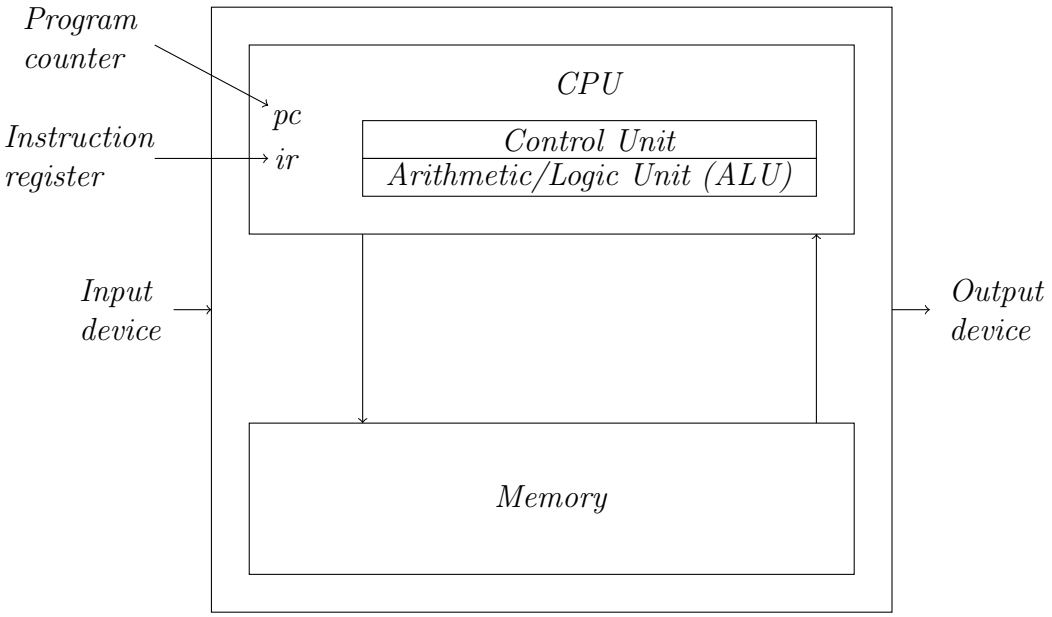

Figure 6: Von Neumann Architecture.

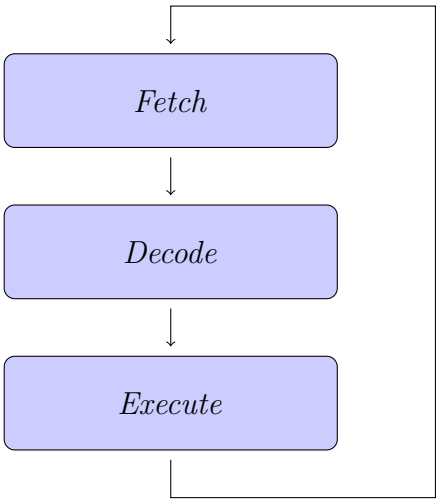

Figure 7: Fetch-Decode-Execute cycle.

DLX Machine

- Control unit:
  Instruction register (ir) and program counter (pc).
- Arithmetic unit:
  32x 32-bit registers; `reg[0]`, `reg[1]`, ..., `reg[31]`. `reg[0]` always contains the value `0` and `reg[31]` is the link register. Both are reserved by convention and must not be used for any other purpose.
- Memory:
  n 32-bit words, byte-addressed (see Figure 8), word-aligned;
  `mem[0]`, ..., `mem[n - 1]`.

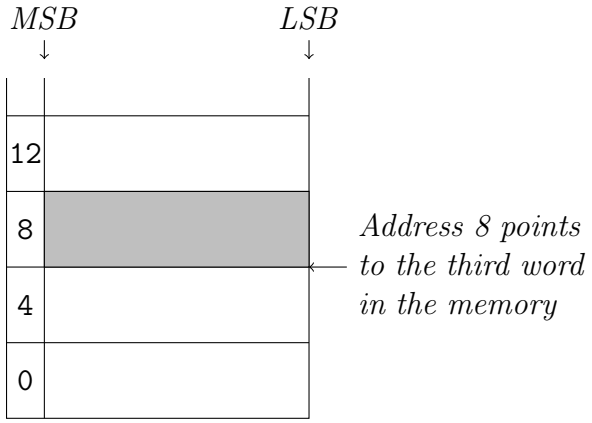

Figure 8: Visualization of a byte-addressed memory of 32-bit words.

Syntax Formats

General syntax of an instruction: `op a, b, c`.

F1

The length of `a` and `b` allow to address all 32 registers. The two's complement is used here because the implementation of arithmetics is easier (in contrast to the one's complement).

op	$0 \leq a \leq 31$	$0 \leq b \leq 31$	$-2^{15} \leq c \leq 2^{15} - 1$
6 bits	*5 bits*	*5 bits*	*16 bits sign-extended to 32 bits*
2^6 *ops*	$2^5 - 1$	$2^5 - 1$	

F2

E.g. `R1 = R2 + R3`

op	$0 \leq a \leq 31$	$0 \leq b \leq 31$	*unused*	$0 \leq c \leq 31$
6 bits	*5 bits*	*5 bits*	*11 bits*	*5 bits*
2^6 *ops*	$2^5 - 1$	$2^5 - 1$		$2^5 - 1$

F3

Absolute addressing in memory.

op	$0 \leq c \leq 2^{26} - 1$
6 bits	*26 bits*
2^6 *ops*	$2^{26} - 1$

Von Neumann Bottleneck: Memory read/write operations limit the performance.

Register Instructions

Arithmetic Instructions

F1

Instruction	Semantics	Additional information
`ADDI a, b, c`	`reg[a]:=reg[b]+c;`	Add immediate
	`pc:=pc+4;`	`c` is data (a constant)
`SUBI a, b, c`	`reg[a]:=reg[b]-c;`	Substract immediate
	`pc:=pc+4;`	`c` is data (a constant)
`MULI a, b, c`	`reg[a]:=reg[b]*c;`	Multiply immediate
	`pc:=pc+4;`	`c` is data (a constant)
`DIVI a, b, c`	`reg[a]:=reg[b]/c;`	Divide immediate
	`pc:=pc+4;`	`c` is data (a constant)
`MODI a, b, c`	`reg[a]:=reg[b]%c;`	Modulo immediate
	`pc:=pc+4;`	`c` is data (a constant)
`CMPI a, b, c`	`reg[a]:=reg[b]-c;`	Compare immediate
	`pc:=pc+4;`	`c` is data (a constant)
		`reg[a]==0 if reg[b]==c`
		`reg[a]>=0 if reg[b]>=c`
		`reg[a]>0 if reg[b]>c`
		`reg[a]=<0 if reg[b]=<c`
		`reg[a]<0 if reg[b]<c`
		`reg[a]!=0 if reg[b]!=c`

F2

Instruction	Semantics	Additional information
`ADD a, b, c`	`reg[a]:=reg[b]+reg[c];`	Add
	`pc:=pc+4;`	Register addressing `reg[c]`
`SUB a, b, c`	`reg[a]:=reg[b]-reg[c];`	Substract
	`pc:=pc+4;`	Register addressing `reg[c]`
`MUL a, b, c`	`reg[a]:=reg[b]*reg[c];`	Multiply
	`pc:=pc+4;`	Register addressing `reg[c]`
`DIV a, b, c`	`reg[a]:=reg[b]/reg[c];`	Divide
	`pc:=pc+4;`	Register addressing `reg[c]`
`MOD a, b, c`	`reg[a]:=reg[b]%reg[c];`	Modulo
	`pc:=pc+4;`	Register addressing `reg[c]`
`CMP a, b, c`	`reg[a]:=reg[b]-reg[c];`	Compare
	`pc:=pc+4;`	Register addressing `reg[c]`
		`reg[a]==0 if reg[b]==reg[c]`
		`reg[a]>=0 if reg[b]>=reg[c]`
		`reg[a]>0 if reg[b]>reg[c]`
		`reg[a]=<0 if reg[b]=<reg[c]`
		`reg[a]<0 if reg[b]<reg[c]`
		`reg[a]!=0 if reg[b]!=reg[c]`

Register Allocation Problem

Registers have to be used in order to execute an instruction. There are 29 registers (in theory) to use. In practice, at least in this course, less registers can be used because some registers are reserved for special purposes (stack pointer, heap pointer, globals pointer, frame pointer, ...). It must be guaranteed that these will not be used and furthermore already allocated registers must not be used for an instruction.

Examples:

C Code	Instructions	Additional information

`1 + 2;`	`ADDI 1, 0, 1`
	`ADDI 2, 0, 2`
	`ADD 1, 1, 2`
		First, naive solution

`1 + 2;`	`ADDI 1, 0, 1` `ADDI 1, 1, 2`	Second, better solution
`1 + 2;`	`ADDI 1, 0, 3`	Third, best solution Constant folding

`if(1 < 2)`	`ADDI 1, 0, 1`
	`ADDI 2, 0, 2`
	`CMP 1, 0, 2`
	`BGE 1, 0, <loc>`
		`<loc>` represents the line of code in the assembly code the CPU continues with

Memory Instructions

F1

Instruction	Semantics	Additional information
`LDW a, b, c`	`reg[a]:=mem[(reg[b]+c)/4];` `pc:=pc+4;`	Load word (from memory) Register-relative addressing
`STW a, b, c`	`mem[(reg[b]+c)/4]:=reg[a];` `pc:=pc+4;`	Store word (into memory) Register-relative addressing

`POP a, b, c`	`reg[a]:=mem[reg[b]/4];`
	`reg[b]:=reg[b]+c;`
	`pc:=pc+4;`
		Pop (from stack)
		`c`: size of the popped chunk
		`reg[b]`: contains stack pointer

`PSH a, b, c`	`reg[b]:=reg[b]-c;`
	`mem[reg[b]/4]:=reg[a];`
	`pc:=pc+4;`
		Push (onto stack)
		`c`: size of the popped chunk
		`reg[b]`: contains stack pointer

The stack grows from high to low addresses (top-down). In the `POP` and `PSH` instructions, `reg[b]:=reg[b]+c` and `reg[b]:=reg[b]-c` represent the actual removal of the element from the stack, respectively. In case of the C* language `c` will always be `4` because only a single type (`int`) exists in this particular language. In general `c` is the amount of data you want to remove from the stack.

Without register-relative addressing, which is a concrete form of indirect addressing, you could only talk about as many memory cells as your program uses. The only way to get the same expressivity as with register-relative addressing is to write self-modifying code.

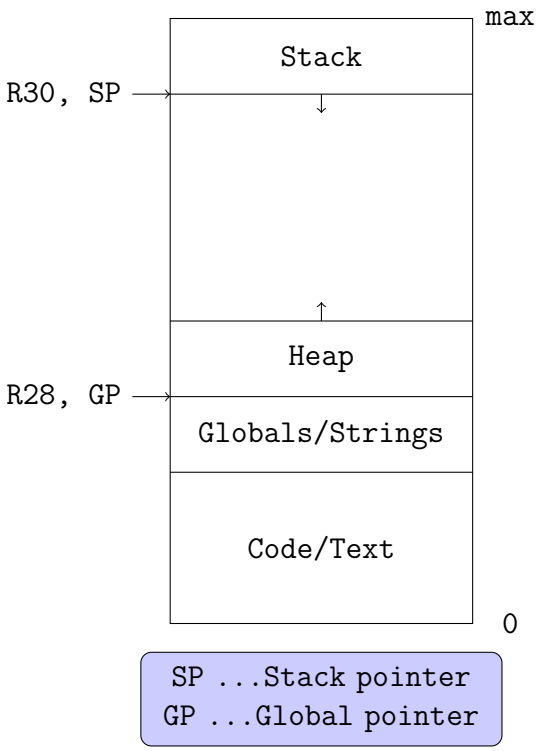

Figure 9: Memory layout when executing a program

Examples:

C Code	Instructions	Additional information
`int x;`	`LDW 1, 28, -4`	`R1 := x`
`x = x + 1;`	`ADDI 2, 0, 1`	`R2 := 1`
	`ADD 1, 1, 2`	`R1 := R1 + R2`
	`STW 1, 28, -4`	`mem[x] = R1`
`*x + 1;`	`LDW 1, 28, -4`	`R1 := x`
	`LDW 1, 1, 0`	`R1 := mem[R1 + 0]`
	`ADDI 2, 0, 1`	`R2 := 1`
	`ADD 1, 1, 2`	`R1 := R1 + R2`
`*(x + 1);`	`LDW 1, 28, -4`	`R1 := x`
	`ADDI 2, 0, 4`	`R2 := 0 + 4`, (4 because it's an address)
	`ADD 1, 1, 2`	`R1 := R1 + R2`
	`LDW 1, 1, 0`	`R1 := mem[R1 + 0]`
		`*(x + i)` is equivalent to `x[i]`

When dealing with `*(x + 1)`, the type of `x` has to be known as well as its size in the memory. The size of `x` in the memory determines the offset which is added to the address of `x`. In case of the C* language this value is always `4` because there is only one type and no support for composed types (`struct`). Nevertheless, the size of the type needs to be added in general.

Formula to compute `*(x + i) = x[i]`: `x` + `sizeof(x)` * `i`.

Control Instructions

F1

Instruction	Semantics	Additional information
`BEQ a, c`	`if(reg[a]==0) pc:=pc+c*4;`	Branch on equal to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BGE a, c`	`if(reg[a]>=0) pc:=pc+c*4;`	Branch on greater than or equal to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BGT a, c`	`if(reg[a]>0) pc:=pc+c*4;`	Branch on greater than to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BLE a, c`	`if(reg[a]<=0) pc:=pc+c*4;`	Branch on less than or equal to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BLT a, c`	`if(reg[a]<0) pc:=pc+c*4;`	Branch on less than to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BNE a, c`	`if(reg[a]!=0) pc:=pc+c*4;`	Branch on not equal to zero
	`else pc:=pc+4;`	Conditional branch, pc-relative
`BR c`	`pc:=pc+c*4;`	Branch (unconditional)
`BSR c`	`reg[31]:=pc+4;`	Branch to subroutine (unconditional)
	`pc:=pc+c*4;`	Tthe link register (`R31`) is saved to be able to return to the correct instruction

F2

Instruction	Semantics	Additional information
`RET c`	`pc:=reg[c];`	`c` = `R31`

F3

Instruction	Semantics	Additional information
`JSR c`	`reg[31]:=pc+4;`	`c` = `R31`
	`pc:=c;`	absolute addressing

All branches (conditional as well as unconditional) are pc-relative. The multiplication of `c` by `4` is because the words in the memory are byte-addressed. Branches are useful (especially because they operate pc-relative) because you generate so-called *relocatable code*. *Relocatable code* can be moved in memory anywhere and will not change its behavior because every address is computed relative to the program counter. The drawback of branches is the restriction of the addressable space: there may be more memory than you can use with branches. The solution is absolute addressing used by the F3 format. The F3 format allows you to address a range of 2^{26}.