FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Tiny x86 - Architecture Simulator for Educational Purposes
Student:	Bc. Ivo Strejc
Supervisor:	Ing. Petr Máj
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

For the purposes of the MI-GEN course, design a simple register virtual machine reminiscent of the x86 architecture (limited number of registers, stack, non-orthogonal instructions of different length) so that the virtual machine can be used as a target for a compiler backend exercising the topics covered in the MI-GEN course (register allocation, instruction scheduling & selection, various optimizations). Implement such virtual machine using the C++ language with attention to easy interface and show its validity by implementing a compiler for the tinyc language to the virtual machine.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 26 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Tiny x86 - Architecture Simulator for Educational Purposes

Bc. Ivo Strejc

Department of Theoretical Computer Science
Supervisor: Ing. Petr Máj

May 6, 2021

Acknowledgements

I would like to thank my supervisor Ing. Petr Máj, for his patience and guidance. I am also grateful for my family and friends support, especially in such challenging times.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 6, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Strejc, Ivo. *Tiny x86 - Architecture Simulator for Educational Purposes.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce prezentuje *tiny x86* architekturu a virtuální stroj, určené jako pomocný nástroj studentům k porozumění technikám kompilování a jejich dopad na výkon programu. V porovnání s již existujícími instrukčními sadami je *tiny x86* jednodušší na použití, protože oproti binárnímu kódování nabízí aplikační rozhraní v jazyce C++ a nelimituje se na jeden návrh (jsou podporovány prvky CISC i RISC architektury). Prezentovaný virtuální stroj nabízí rozsáhle možnosti konfigurace, dovolující z(ne)výraznit různé návrhové prvky (počet registrů, odezvu paměti, trvání instrukcí atd.). Virtuální stroj je již nasazen v předmětu NI-GEN (generování kódu) na FIT ČVUT, kde jeho jednoduchost dovoluje studentům během semestru psát kompletní kompilátor.

Klíčová slova VM, kompilátory, ISA, tiny x86, tinyverse

Abstract

This thesis presents *tiny x86* architecture and virtual machine designed to help students understand various compiler techniques and their effect on the program performance. Compared to existing instruction set architectures, *tiny x86* is simpler, easier to use as it comes with a C++ API as opposed to binary encodings and does not limit itself to single design principles (both CISC and RISC features are supported). The VM also offers extensive configuration options, allowing it to (de-)emphasize various architecture features (register pressure, memory latency, instruction timings, etc.). The VM is already used in the NI-GEN (Code Generation) course at FIT CTU, where its simplicity allows the students to write full compiler pipeline during the term.

Keywords VM, compilers, ISA, tiny x86, tinyverse

Contents

List of Figures

Introduction

Most code programmers write today is in programming languages, that aim for human readability and expressiveness. But processors do not understand these higher languages, made for humans and our way of thinking. A processor understands only its lower level language - instructions, that humans tend to find hard to read since their extreme level of detail and verbosity obscures the algorithm itself.
This gap necessitates some sort of translation. This can be done either at runtime (while the program is executed) using interpreters, or statically (before the program is executed) using compilers. Interpreters executes program indirectly: they take the source program statement by statement, and call a corresponding routine. This comes at a cost as the code cannot be optimized, but an interpreter can start execution almost instantly. Code of interpreted languages can be executed on any platform that has implementation of such interpreter. Compilers on the other hand translate given code straight to the native code of the target. This machine code can then be executed many times without any additional cost. Such compiled programs, running directly on processor, usually perform better than programs written in interpreted languages [1]. Compilers are usually separated into two parts - frontend, accommodating parsing of the source code, and backend, providing optimizations and translation to the target language [2]. Depending on the target, different translation and optimization approaches and techniques are used. Low-level, platform specific, techniques like register allocation and instruction selection are crucial for a compiler that produces performant machine code.
As a part of Systems programming program taught at FIT CTU a course NI-GEN (Code Generation) focuses on compilers. Different techniques for compilation of modern programming languages are practiced, with emphasis on explaining how to write a compiler as a whole, not focusing on specific platforms or languages. The course uses a simple programming language, *tinyC*, a subset of C. tinyC was created to keep the interesting parts of C, like pointers, arrays and structures, without having to worry about other technical details,

that are important for real world use, but irrelevant for educational purposes (such as macros, multitude of integer datatypes, redundant control flow statements, etc.).
Students implement their own optimizing compiler backend as their semestral project, showcasing understanding of taught topics. Because multiple different architectural designs are explored, if real architectures like x86, ARM, MIPS or RISC-V would be used, students would have to switch from one to another based on currently exercised technique. Students would also have to study given targets specifics, interfaces and VMs, which by itself could be a standalone course. As an example, x86, one of the most common architectures, has hundreds of instructions, with extremely complex instruction encoding. It provides only handful of registers, some of which play specific roles in certain instructions. Being of CISC heritage, many of these instructions have narrow use-case, performing very specific and complex tasks. Take the LEA instruction as an example, whose purpose is to calculate a memory address for a higher-language element. Knowing this instruction and being able to use it in such cases can boost program performance by a lot. But other processor designs, such as RISC (ARM, MIPS) opt for more registers and simpler instructions, more of which are necessary to provide the functionality similar to that of the LEA instruction. The architecture selection thus plays a vital role as some aspects of code compilation can only be experienced on certain architectures.
Furthermore, as writing even a simple compiler is a complex undertaking, it would be beneficial for students to be able to abstract from some of the more complex issues to get a minimal viable compiler first, being able to run code as soon as possible and the refine their design by making it work with more and more realistic features. This is something hardly achievable with existing architectures, that are made for hardware CPU implementations (for instance it is almost impossible to write a proper compiler without register allocation or complex instruction encoding schemes).
This creates a need for a virtual target, that can be run on any personal computer for students to experiment with and use as target in their compilers. Such target does not have to deal with problems that architectures running on real processors have to, like efficient encoding, allowing greater freedom in extending and changing the architecture. In this thesis, *tiny x86* architecture, based on widely adopted x86 architecture [3], is introduced. The goal is to create single configurable target for educational purposes, that can be used in during NI-GEN course as compilation target, practicing different conditions and architecture features, while keeping simple, unified user-friendly interface. It resembles x86 architecture in most basic aspects, but it combines many other architectural designs, that can be configured to suit specific needs exercising different techniques.

Chapter overview

The rest of this thesis is organized in the following sections:
Current solutions takes a look at current solutions and discusses their possible usage in NI-GEN course.
CPU design overview focuses on different CPU techniques, that are being used.
Tiny x86 ISA presents designed architecture.
Tiny x86 VM introduces virtual machine implementing *tiny x86* ISA.
Realization describes implementation details of *tiny x86* VM and supporting constructs.
Evaluation looks into output of this thesis and how it was validated.
Conclusion concludes the thesis and offers hints for future improvements.

Chapter 1

Current solutions

In this chapter, some of existing virtual machines are presented. Their intended use-case is analyzed and compared with needs of NI-GEN course and possibly what alteration would be required to suit the use-case.

1.1 Selfie

Selfie is an educational project of the Computational Systems Group at the Department of Computer Sciences of the University of Salzburg in Austria [4]. It provides RISC-U, an easy-to-teach subset of RISC-V, emulator and self-compiling compiler to this architecture for subset of C called C* [5]. Interestingly, *Selfie* is also written in C*, allowing full self hosting. This lead to creation of self-hosted hypervisor, which enters the territory of writing operating systems.
Selfie provides performance counting, that can help students identify bottlenecks and benchmarking tools.
Selfie and this thesis share one common goal - create educational project used for teaching compilers, but it does not aim to showcase different techniques used on different architectures, rather than self-hosting. Given *Selfie*s maturity and interesting use-cases, it now serves as much universal tool, a feat to be desired by *tiny x86*.

1.2 QEMU

QEMU, short for **Q**uick **EMU**lator, is a generic and open source machine emulator supporting multiple hardware platforms including x86, ARM, PowerPC, MIPS and RISCV [6].
QEMU allows to easily run x86 programs by providing boot sector image containing your program. This has few limitations - mainly to even begin with, you have to prepare your VM by setting data segments etc., bringing additional complexity. Another limitation is the boot sector max size, which is 512 bytes. This might be enough for basic un-optimized programs, but when translating more complicated programs, compiler producing size un-optimized code could run out of space, causing parts of the code not being loaded.
Other ways how to run user program would be creating custom kernel, but then you have to take care of multiple things, for example interrupts, multiprocessing, console input/output.
QEMU provides system configuration, but it is build around specific ISAs in code, so changing these would require deep code knowledge and patching of the source code. No standard way of performance counting and its processing is built into QEMU, so this would also require extending the source code. From this I conclude, that this project is suitable for production. Regarding educational purposes, it could be useful if the study subject is either x86 specifically, or creating OS using some already existing compiler.

1.3 LLVM

LLVM is a collection of compiler and toolchain technologies. It began as a research project at the University of Illinois as a compiler framework [7]. It has grown since to house many subprojects like *Clang* (C/C++ compiler) [8] and *LLVM core* library build around low-level code representation LLVM IR [9], an intermediate representation used in target-independent optimizations and code-generation for many CPUs[10].
LLVM focuses on many fields, but regarding targets, no part seems really viable. LLVM IR is available, which can be executed separately, but does not allow any further control over the architecture without deep codebase knowledge and patching, working more like a scripting language. It also provides no performance counting. Still, usage of LLVM might be something interesting, especially trying to combine LLVM IR as backend with *tiny x86* as target.

Chapter 2

CPU design overview

This chapter focuses on different aspects of CPU design. Different instruction set architecture designs are discussed and examples of existing ones are presented. Next, different processor techniques used to increase performance are described.

2.1 ISA

Instruction set architecture describes CPU on an abstract level. It defines instructions, addressing modes, encoding and how many registers of what kind CPU has. It can specify how memory is accessed, input and output model and special register purposes.
Different CPU implementing this ISA should all be able to run the same machine code. Each of this CPUs can have different performance, efficiency or cost, while producing the same results. This enables us users change processors of the same architecture without needing to change or recompile software that ran on the old CPU.
ISA can be extended overtime, keeping backwards compatibility, but any change to the old design would cause incompatibility. Careful design is required for long lasting ISA.
Many decisions have to be made during design ISA, such as defining more simpler, but possibly less performative instructions, or very broad, rich instruction set, that is hard to master. A lot of care has to be put into instruction encoding, for both large enought different instruction encoding, and fast and precise decoding.

2.1.1 x86

Intel's x86 ISA originates from Intel 3086 processor released in 1978 [11]. Current Intel and AMD processors use extended x86_64 ISA, but x86 is the main focus of this section.
x86 is considered to be CISC design with variable length instructions [12, 13]. The fact, they are able to keep compatibility with processors released over 40 years shows, how much effort is put into its design. Intel's engineers achieved this by using several techniques, like instruction prefixes, altering following instruction's meaning.
But its historical compatibility brings few aspects, that might not make sense in modern environment. Example of this is segmentation. This allows program to set offset to all memory accesses - this was very useful when registers very only 16bit and memory was larger than 65K. From modern stand point, where modern x86 specifies 32bit registers and for x86_64 even 64bit ones, this seems really unnecessary and creating another layer of complexity to writing x86 programs.
x86 provides only a handful of full 32-bit registers that can be broken down to 16-bit or even 8-bit registers. Many of these registers also serve a specific purpose in the ISA. Example of this would be register EAX, that often times serves as the accumulator for instructions. Also, some instructions support only specific addressing modes, which can force compilers into smart instruction selection.
In this architecture, many instructions with very specific use-cases are defined. Understanding them requires deep knowledge, creating high barrier of entry for new optimizing compiler writers.

2.1.2 ARM

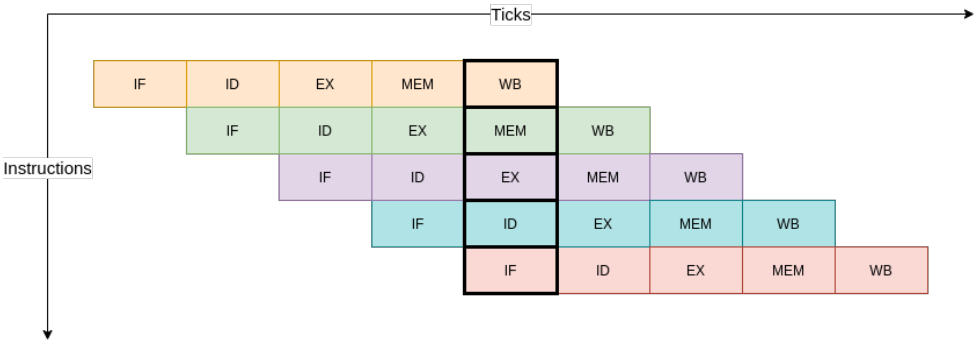
ARM is a family of RISC processors introduced in 1985. Due to its reduced cost, power consumption and thermal output it is used in battery-powered devices like phones, watches or tablets, but is also used in personal computers or servers. The fastest supercomputer at the moment, Fugaku, runs on ARM architecture [14].
There have been several generations of ARM design and both 32-bit and 64-bit versions are available. In RISC fashion, it provides concise number of instruction with limited addressing modes. Typical for this family of instructions, many operations explicitly specify destination register, for example *ADD R0 R1 1*, meaning $R0 := R1+1$, is not available in architectures like x86.

2.2 Scalar processors

Scalar processors are the simplest class of processors. Every instruction is executed strictly in order, in which it occurs. Some techniques can be deployed to increase the throughput of such processor. Some of these techniques are presented in following sections.

2.2.1 Pipelining

Pipelining utilizes that some of the steps in instruction execution can be done in parallel for multiple instructions. For example when while one instruction is being decoded, some other could write to memory. Pipeline, created from several of these distinct stages, is created, processing several instructions at once.
For MIPS32, a RISC processor, these stages are *Instruction Fetch*, *Instruction Decode*, *Execute*, *Memory access* and *Write back* [15]. Many other RISC processor follow very similar partitioning.
These stages can work in parallel to each other and, given ideal conditions, every tick each instruction advances to the next pipeline stage. Because every tick an instruction would leave the pipeline, meaning its execution has concluded, there is only one tick delay between previous instruction.



2.2.1.1 Hazards

Hazards identify situations in integrity of the execution could be disturbed. In some of these causes, only solution is stalling progression of some pipeline stages. All pipeline hazards must be detected and resolved in order for the processor to correctly execute programs.
Data hazards occurs, when instruction dependencies can be violated. Typically these data hazards are recognized:

- Read after write (RAW)
- Write after read (WAR)
- Write after write (WAW)

Control hazards can be viewed as a form of RAW hazard on PC caused by branch instruction. If this is not resolved, incorrect instructions would be executed.
Structural hazards occur, when the CPU has some limited resources, that cannot be shared, and two or more instructions in the pipeline need this resource.

2.2.2 Speculative execution

To avoid stalling in pipelining processor when branch instruction is met, CPU starts speculatively execute following parts of the program, as if no branch would happen. If branching happens, the CPU has to make sure, that no side-effects of this speculative execution happen, clearing the affected parts of pipeline and undo any other effects.
To achieve the best throughput, wrong speculation should be avoided, as they are heavily penalized. This is where branch predictors come into a play. These predict if branching happens and what will be destination. A lot of research is done in this department, as it can greatly improve processors improvement [16]. Studies are being done on advanced branch prediction and how they can be deployed [17].

2.2.3 Register Renaming

In processors, number of physical registers can be larger than number of logical registers - those registers that ISA defines. Logical register than can be mapped to those physical registers. CPU designers can use this fact to their benefit as changing of this mapping, called register renaming, can eliminate stall caused by fake data dependencies, that would otherwise be identified as data hazards. When instruction is being decoded, the renaming takes place. For each register, that need to be read, appropriate physical register is substituted and for register writes, an unused register is selected and noted for future translations.

```
ADD R0 R1 R2 -> ADD PR2 PR3 PR1
```

Listing 2.1: Example of register renaming

To keep track of the mapping and used registers, register allocation table is used. Depending on overall design, the complexity of this table varies, but simplest implementation can be an array, where indexes represent logical registers and the value of given element is physical register.

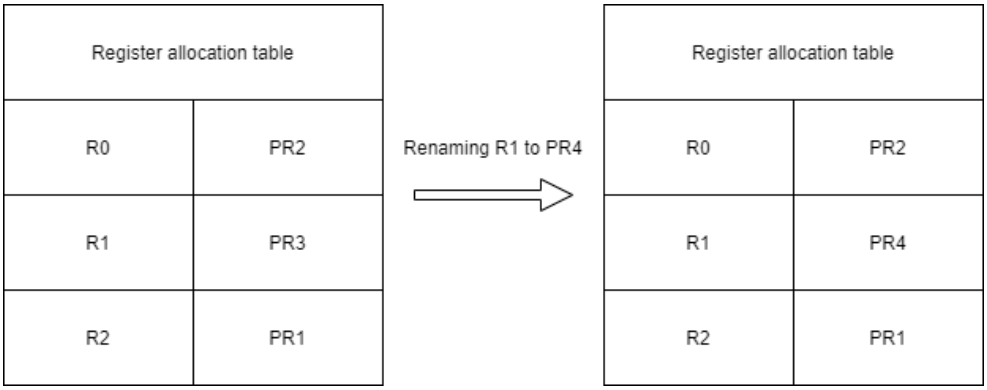


Figure 2.1: Register allocation table and renaming

2.3 Superscalar processors

Superscalar processors can execute multiple instruction per clock cycle, using different techniques. In this section, some of these techniques are described with emphasis put on out-of-order execution paradigm.
Unlike vector processors, executing same instruction over multiple data items, superscalar processors can execute different instructions. Simplest transition from scalar to superscalar processor would be multiplying the pipeline.

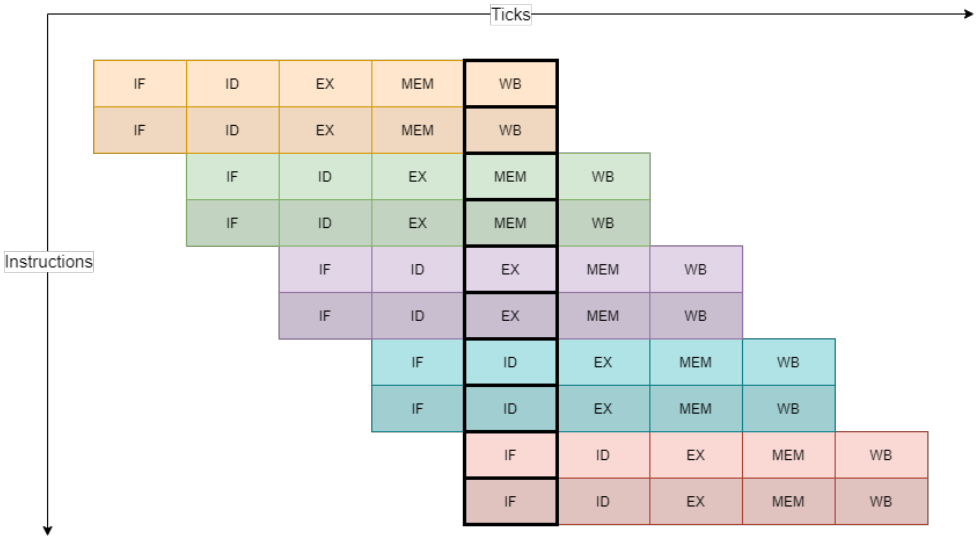


Figure 2.2: Superscalar pipeline

2.3.1 The Classic Tomasulo Algorithm

Designed for IBM 360/91's floating-point unit, Tomasulo's algorithm was a predecessor to modern superscalar processor design. One of the key features this algorithm introduced was its register data flow techniques [18].
Tomasulo's algorithm adds three new mechanisms to the original FPU design - *reservation stations*, the *common data bus* and *register tags*. In original FPU design, each functional unit could hold only one instruction inside its buffer on input side. This meant issuing of instruction to busy unit would cause stalling. To alleviate this bottlenect, reservation station were introduced, replacing the single buffer on input side. From the point of view of FLOS, these stations are viewed as virtual functional units, so as long there is a free reservation station the FLOS can issue an instruction even if the actual functional unit the station is connected to is busy. With the instruction of reservation stations FLOS can now issue instruction even if all of their operands are not fetched. The instruction will wait in the reservation station for all of their operands and only after all of them are fetch instruction becomes ready for execution.
Common data bus (CBD) connects outputs of functional units to the reservation stations. Once functional unit produces a result, the result is broadcasted into the CBD. Instructions in reservation stations needing these results as their operands, latch in the data from CBD.
Register renaming is used to resolve RAW and WAR hazards, eliminating pipeline stalls. To signal, that register is not prepared for reading, a busy flag is introduced.

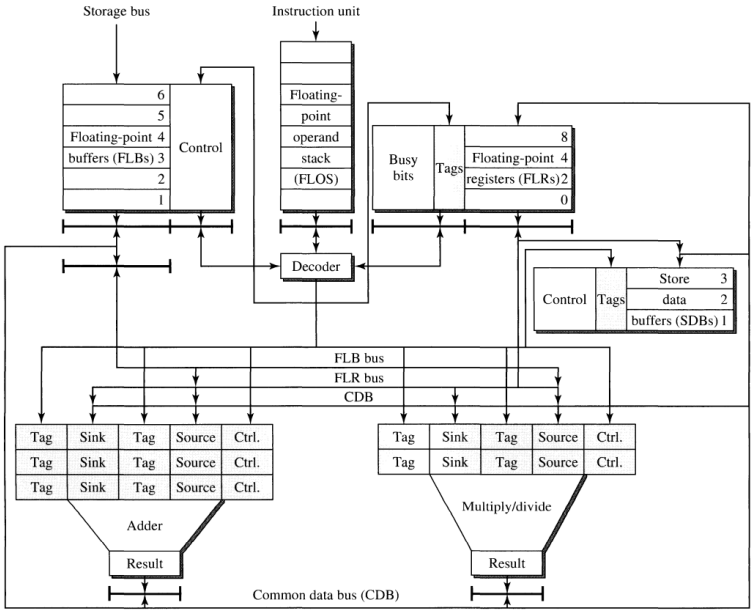


Figure 2.3: Tomasulo design

2.3.2 Dynamic Execution Core

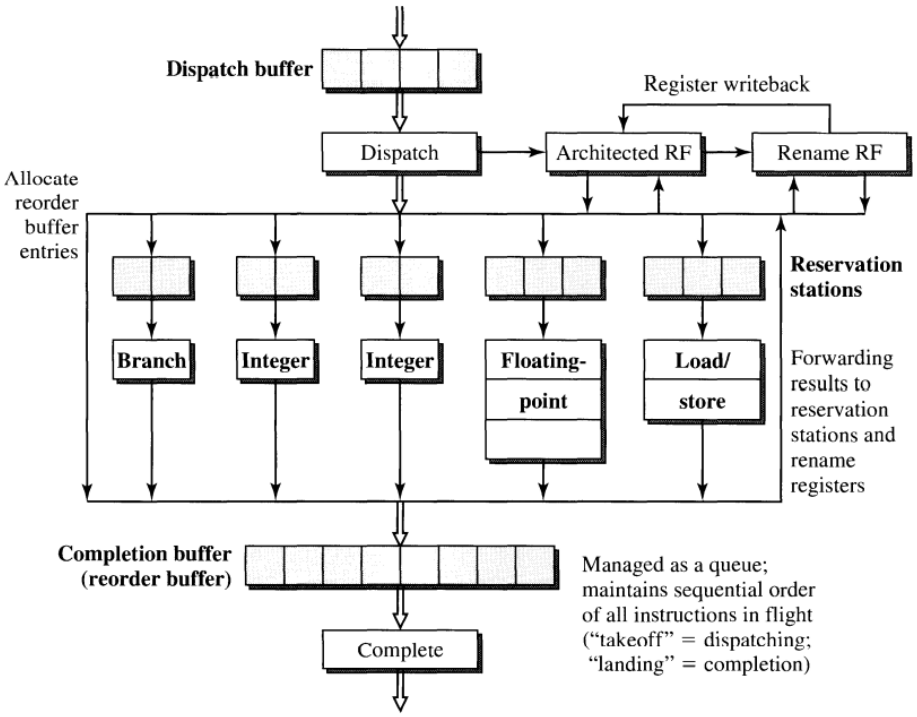


Figure 2.4: Dynamic execution core

Current state-of-the-art superscalar processors utilize out-of-order execution core put in between in-order front end, containing of fetch and dispatch stages, and in-order back end, which completes and retires instructions in program order. This technique resembles a refined Tomasulo's algorithm [19].
Whole execution process can be split into 3 main stages - instruction dispatching, instruction execution and instruction completion.
Instruction dispatching stage consist of renaming of destination registers, allocating reservation station and reorder buffer entries. If allocation is successful, instruction is advanced from dispatch buffer to the reservation station.
Instruction execution stage provides following functionality - issuing ready instructions, executing issued instructions and forwarding results. When instruction fist arrives into this stage, not all of its operands might be fetched. It has to wait for all of its operands to be issued for execution. Similarly as in Tomasulo's design, instructions waiting for issuing listen on available busses for specific tags of their operands, fetching them as they occur. Once instruction is issued into functional unit, an instruction is executed and no further stalls in this stage are expected. Once execution finished and result is

available, it is broadcasted to the forwarding bus alongside its identifier.
Instruction completion takes place in the reorder buffer. Reorder buffer ensures that instructions are architecturally completed in correct order. It can also take care of potential branch predictions misses by keeping track of what branch caused the instruction to be executed and if that branch prediction was miss not completing given instruction.

2.4 Other processor techniques

Many other techniques exist and are being deployed in modern processors, but are really advanced and have limited benefit for use-case of this thesis, yet worth mentioning.
VLIW (very long instruction word) defines such instructions, that perform several predefined calculations in parallel. To utilize this parallelism, compilers have to adjust to such architecture.
SIMD (singe instruction multiple data) is architecture, defining instructions working with array of data at once. This can greatly improve performance for certain workloads, like matrix manipulation. Modern GPUs implement such architecture.
EPIC (explicitly parallel instruction computing) is computing paradigm, that builds on VLIW, improving on several aspects and even creating so called instruction bundles, that define its dependencies and can be issued in parallel[20]. Again, most of the burden of parallelism is put on compilers.

Chapter 3

Tiny x86 ISA

This chapter takes a loot at the ISA that was created for Tiny x86 target. First few key design choices and their implications are discussed. Then I introduce memory model, available registers and addressing modes, followed by instruction descriptions.
Because this is supposed to be educational ISA, many things need to be configurable, to showcase what approaches are more suitable for different platforms. This makes this ISA more fluid, that others usually are. What I define in terms of possible registers, addressing modes and instructions is the superset of available configurations.
Very important aspect of this ISA is that the resulting program does not have to be stored into a file or any other binary representation. Also no linking will be done, as only single compilation units of *tinyC* programs are compiled. It is intended to be stored in memory the whole process in arbitrary representation, from input program compilation to the actual execution. This strips the need for encoding instructions and its operands. But because instruction length has influence on execution performance, each specific configuration can allow different instruction length for each instruction and its operands.

3.1 Memory model

tiny x86 uses Harvard architecture, meaning instructions are separated from data. Instructions accessing memory can only alter data and not program it self.
Data is aligned to 64-bit blocks and addressable by index of such block. Any finer addressing has to be done on program level.
Memory includes stack, growing down from the end, and loaded data from program growing up beginning. No paging or other segmentation is defined.

3.2 Registers

tiny x86 introduces few quite standard registers that can be found in most of modern processors:

- **PC** - program counter, indicating position in program
- **SP** - stack pointer
- **BP** - base pointer, pointing to beginning of current function stack frame
- **FLAGS** - flags set by ALU operations

Except for base pointer, all of these register play very specific roles in instruction. Base pointer is an exception, because it server mostly compilers to work with function stack frames.
Further more, *tiny86* specifies other registers - these are fully general purpose, their usage is fully determined by the compiler. *tiny86* defines two types of registers, integer and float registers. Both register types are 64bit long, allowing easy value conversion and manipulation. I still decided to distinguish integer and float registers for clarity of generated code and possibly prevent bugs by not allowing interchangabilily.
Number of registers, both integer and float, is not fixed. It is expected to be configurable, depending on VM. Integer registers are denoted as *Ri*, where i is index of given register. Similarly, float registers are denoted as *FRi*. As is programming tradition, registers are indexed from 0.
For simplicity, from now on if not stated otherwise in this thesis, when I use the term register, integer register is meant. Also every previosly mentioned special registers hold integer values and can be treated as integer registers.

3.3 Addressing modes

Addressing modes specify, what kind of operands instructions can have. In this section addressing modes are specified, with examples where appropriate. Note that not every instruction has to work with given addressing mode. This is specified by the instruction it self.
Immediate values:

- Integer immediate
- Float immediate

Register based:

- Register - *R0*
- Float register - *FR0*
- Register offset - *R0 + 10*

Some other register based modes are available, but not expected not to be used like directly in instruction operands. They were created, because their definition makes the memory accesses easier and they remain as a possibility for future instructions. **Memory accesses**:

- Memory immediate - *[10]*
- Memory register - *[R0]*
- Memory register offset - *[R0 + 10]*
- Memory register and register - *[R0 + R1]*
- Memory scaled register - *[R0 * 2]*
- Memory register offset and register - *[R0 + 10 + R1]*
- Memory register and scaled register - *[R0 + R1 * 2]*
- Memory register offset and scaled register - *[R0 + 10 + R1 * 2]*

Some of these modes exist because of specific use-cases defined by standard programming constructs. For example $[R0 + R1 * 4]$ could be used to access array elements. $R0$ indicated starting address of the array, $R1$ represents the index and is multiplyed by the size of stored element.

3.4 Instructions

This section highlights different categories of instructions, that *tiny x86* defines, and what addressing modes they support. Full list of individual instructions with detailed description is included in Appendix C.

Moving - *MOV* instruction, that serves for moving data from and to registers and memory. Addressing modes available are registers, values and every memory accessing. Moving between float and integer registers binary copies value without any narrowing or extending.

Integer arithmetics and bit operations - instruction for standard mathematical operations, including bit manipulation, utilizing ALU. Both binary and unary operations are present. These operations happen on register, and if applicable, second operand can be either value, register or register offset. RISC like version are available as well, for example *ADD R0 R1 1*.

Control flow category concists of conditional and unconditional jumps. Conditional jumps decide based on *FLAGS* register and should be after *CMP* instruction, comparing register to some selected operands. Jump destination can be defined by address (immediate value) or register.

Functional are instruction enabling function calling - *CALL* and *RET*. *CALL* can specify its target either by immediate value (an address) or by register, allowing for higher-order functions using function pointers.

Stack manipulation is done using two quite standard instructions *PUSH* and *POP*.

IO - defines *PUTCHAR* and *GETCHAR*, printing to the output of the VM. This allows students to quickly see results of their programs that is otherwise done using underlying operating system running given program.

Float arithmetics - basic float arithmetics are defined, with similar to integer based ones, with *F* prefix and set the same *FLAGS* register and can be used as condition for conditional jumps. *NRW* for narrowing from float to integer and *EXT* for extending integers to floats are defined.

VM manipulation - *DBG* and *BREAK* are defined for debugging purposes, *HALT* for halting the VM, otherwise infinite execution of *NOP* (no operation) happens. *CLR* for clearing flags is defined.

Others - *LEA* for obtaining address using advanced memory addressing modes is included, as well as already mentioned *CMP* and its float variant *FCMP* for comparing two values, usually followed by conditional jump instructions.

Chapter 4

Tiny x86 VM

This chapter describes the designed virtual machine, implementing *tiny x86* ISA. After reading this chapter, users of this VM should have notion of how program is executed and how different situations can influence performace of such program.
First, memory design is presented. Then CPU design and used processor techniques are described.

4.1 RAM

Configurable random access memory with simple interface for use in *tiny x86* VM is defined in this section.
Because *tiny x86* ISA specifies, that Harvard architecture is used, we really have to worry only about defining memory for the data. Instruction will be stored separately managed by processor.
Total size is fully customizable, but some limits should be considered, especially if any function calls should take place. As defined by the ISA, data accesses are aligned to 64-bit values. Size of RAM will be specified as number of these 64-bit values, that can be stored, meaning if RAM of size 1024 is defined, its actuall physical size will be $1024 * 8$ bytes.
Memory has also configurable number of read gates. Multiple gates can perform reads at the same time, but no reads should happen without assigned gate. If multiple requests for single address are issued, only one gate is used. Read from memory address that is being written to is undefined behavior, forcing CPU to keep track of ongoing writes.

4.2 CPU

This section describes design of processor implementing out-of-order execution engine similar to the one described in subsection 2.3.2. To simulate the limits of real used architectures, like minimizing dependencies between registers or branching, *tiny x86* CPU will implement pipelining with following 5 stages:

- **IF** *Instruction Fetch*
- **ID** *Instruction Decode*
- **OF** *Operand Fetch*
- **EX** *Execution*
- **RET** *Retirement*

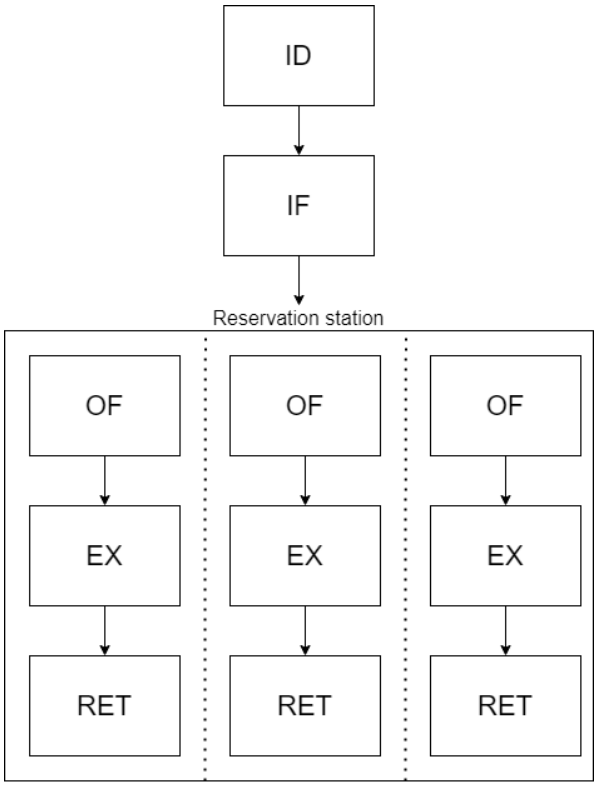


Figure 4.1: Pipeline and reservation station for *tiny x86* VM

Next, out-of-order execution will be used. This is achieved using *reservation station*, based on the design of dynamic execution core, where last 3 stages of the pipeline happen. Reservation station is made of entries, that facilitate instructions, operands and other information required for operand fetching,

executing and retirement. Reservation station also houses multiple ALUs, that can be assigned to an entry if execution requires it.
Each entry has its state from following list:

- Preparing - operands fetching
- Ready - all operands are fetched and as soon as there is a free ALU execution will begin
- Executing
- Retiring - waiting for retirement

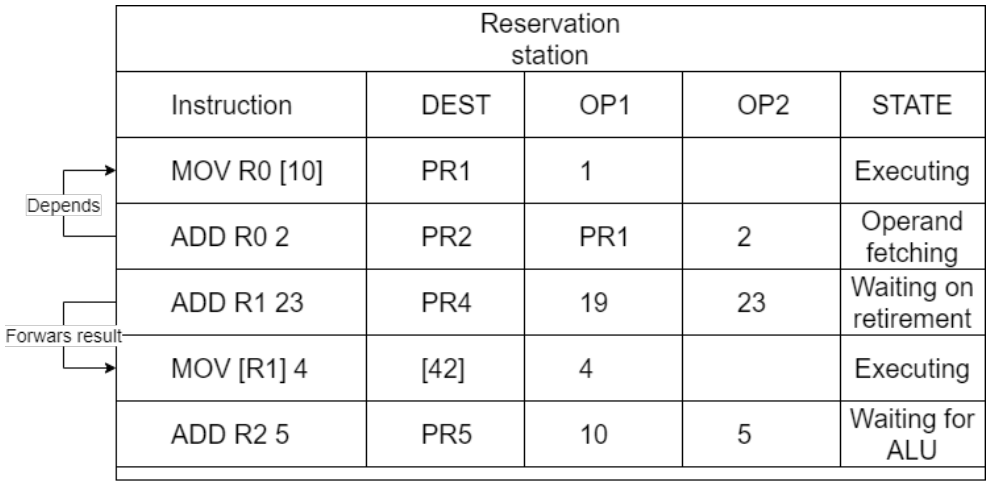


Figure 4.2: Example of reservation station entries

Inclusion of out-of-order execution should serve as a demonstration of power of dynamic scheduling, that can greatly improve compiled programs that did not utilize instruction scheduling techniques.
Number of reservation station entries can be configured, where in-order execution can be easily simulated by setting only one reservation station entry. Number of ALU (arithmetic logic unit) can be configured as well. All ALUs are general purpose, meaning no separation for floating point arithmetic or other specific use-cases is made.

4.2.1 Instruction Fetch

Fetches next instruction. If fetched instruction was a branch instruction, branch prediction is applied.

4.2.2 Instruction Decode

Contrary to its name, there is no actual decoding being done in this stage, the name was chosen to model real processors. Register renaming and memory write registration is done at this stage.

4.2.3 Operand fetch

Fetches all of required operands. This stage, unlike previous stages, can take multiple ticks depending on the execution context. This can be caused by sever things. First one, and probably the most obvious one is reading from memory. Memory has its latency, meaning reading from it can take several ticks. Another possible stall in operand fetching can be dependency on precious instruction. Until that instruction finishes its execution, this operand cannot be fetched. If instruction in this stage requires an ALU, it cannot progress to the next stage, without available ALU.

4.2.4 Execution

In this stage, several things take place - computation using assigned ALU, setting registers including program counter and flags, and setting memory write address and value if required.

4.2.5 Instruction retirement

Presence of this stage and rule, that instruction can retire only, when there are no previous instructions in non-retiring state, ensures, that instruction's side effects, that cannot be simply reversed, happen in order in which they occur in the program. This stage is also important because of the speculative execution, as branch misprediction could influence observable state of the VM. In this stage, memory writes are initialized, branch decisions are processed, possibly enforcing speculative execution unrolling when branch misprediction happened, handling debug and break functions and halting the VM.

4.2.6 Register renaming

To fully utilize out-of-order execution, elimination of false data dependencies is crucial. Register renaming is used to eliminate these hazards and to make tracking of actual hazards easier.
Every time instruction is being added to the reservation station after the instruction decode phase, registers that are possibly affected by execution of given instruction are renamed. This renaming maps logical registers (those, that ISA presents, both integer and float) to some internal register. These internal registers are called physical registers. There physical registers are not addressable from the generated program, only through the translation from logical register. CPU tracks for each physical register if it is ready - physical register becomes ready after a value has been written to it until this physical register is used for renaming again, then the register value becomes unavailable again. By this, we can avoid reading invalid data before they are ready.
These mappings are stored in **RAT** - *register allocation table*. CPU has one main and most updated RAT and each reservation station holds two copies of RATs - one for register read translations, that was copied before any registers were renamed, and one for register write translations, that was copied after all affected registers were renamed. By this coping of RATs we do not have to make difficult bookkeeping of what mapping should what entry use, even though this potentially uses more memory, that would be needed, as both RATs share most of the information.
When speculative execution unrolling happens, the write RAT of the entry that caused the unrolling will be used as the new main CPU RAT. To be able to determine what physical register should be selected for renaming, CPU also counts how many reads/writes are subscribed/queued for given register. Selected physical register has to satisfy these two conditions:

1. There are no subscribed reads/writes to this physical register.
2. There is no mapping to this register in the current RAT.

First condition ensures, that any instruction already added to reservation station will not read or write this physical register. The second one then ensure that all logical registers are mapped to something, in case read from these registers will be required in future.

4.2.7 Memory IO

Due to various aspects of this design, CPU has to manage its memory IO. Again, we have to take care of potential data hazards, especially due to RAM's latency on reads and writes.
Most important thing CPU has to note are memory writes, both ongoing and pending (future). Each of these writes has internal ID. ID is assigned in chronological order, meaning writes with lower ID happened before (from the architectural point of view) writes with higher ID.
Pending writes are writes, that have not yet started writing to RAM. They can specify two things - address and value. If both values are specified, the RAM write can commence during instruction retirement. When instruction is being added to reservation station, CPU collects its future memory writes. There write can already have address and value specified, but they most likely lack some of these and they are specified in later stages of the pipeline.
Ongoing writes are those writes, that began writing to RAM, but are not finished yet. Actual RAM write is started only during instruction retirement. Because of how retirement order is designed, we don't have to worry about canceling or reverting memory write, keeping RAM in valid state at all times. When instruction is added to the reservation station, its entry is assigned the highest registered write ID. This is used during operand fetching.
When operand requires memory read, CPU first checks, if there is any ongoing RAM read to same address and either forwards fetched value (in case the read finished in this particular tick) or stall. If no such read is present, CPU check for possible memory writes, that would prevent from reading RAM. This is determined by both requested address and the aforementioned highest memory write ID - lets call this max ID. We have to find memory write to either unspecified address or address matching requested address with lower or equal ID to max ID. If there is no such write, RAM read will be queued. Otherwise we check if the target address matches requested address and if the write value is specified. In that case, we can forward the value, else stall occurs.
One crucial thing that current CPU design is missing is memory caching. Due to time constraints it is not present and is part of future work.

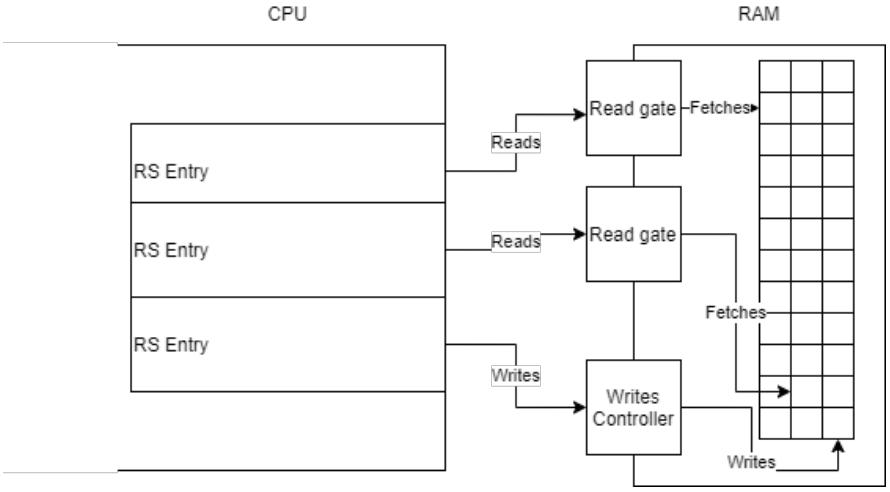


Figure 4.3: Memory IO

4.3 Reporting system

To give students feedback on their compiler, some reporting system has to be created. Resulting reports will be used to measure how well optimized the compiler is, as measuring by actual runtime on host machine has unwanted aspect, as discussed in motivation.

As basis for most statistics calculation is counting how many ticks have happened before the VM halted and how many instruction were executed. Just the first statistic alone can be sufficient for simple benchmarking, combined with throughput, calculated as $\frac{\#\text{ of instuctions executed}}{\#\text{ of ticks}}$, we can start comparing different compiler implementations. But to give students valuable feedback, the reporting system should for each also track stalls, what caused them and for how many ticks did it stall.

Tracked stall correspond to pipeline stall with extra detail of their cause. For instruction fetch and instruction decode stalls happen when following stage is full. For operand fetching, each operand fetch requirement is tracked separately. Another stall for instruction in operand fetching might be unavailable free ALU if given instruction requires it for its execution. In execution part, no stall should happen. For instruction in retirement part, stalls happen, when there are previous instruction that are not in retirement phase.

This tracking should be done for each instruction individually and once consolidated, they can be presented for each instruction signature. Instruction signature consist of instruction type, for example *MOV*, and its operand addressing modes, for example *Reg, Imm*. Result of this report should give the student basic tool to identify bottlenecks of the program their compiler generates and compare different strategies of mitigating these stalls.

Chapter 5

Realization

Tiny x86 is a part of *tinyverse*, project started by this thesis supervisor, Ing. Petr Máj, written in C++. This project houses other components used in NI-GEN course, especially tinyC specification and parser. It also provides basic utility library for logging, printing, testing and configuration management. Tiny x86 is first of possibly many other targets, that students could use, study, modify or extend. This puts few constraints on the implementation. To interconnect with the rest of the project, C++ is obvious choice. At FIT CTU, C/C++ is taught in first year of bachelors study, therefore it is expected for masters students to be able to understand and write C++ code. But still, there is a question of different C++ standards and paradigms.
C++ is still actively in development, giving us new standard approximately every 3 years, with varying delay for compiler vendors to implement new features. At the time of writing this thesis, C++20 is the newest standard, but many compilers do not support this standard fully. Next, mostly supported standard is C++17, that had some time to mature, and its features starts to show up in modern code bases. So C++17 was chosen, but because students have to be kept in mind, only the most useful new features, for example *std::variant* or *std::optional* from STL, were used. Also, hand to hand with this is template usage. Templates in C++ are very powerful tool, but it brings some obscurity to the codebase. Therefore minimal template usage was decided, even if that meant more handwritten code.
Regarding programming paradigm, object oriented programming (OOP) was chosen, as curriculum at FIT CTU contains OOP in several of its courses and is generally considered to be the industry norm. Usage of OOP allows for extensible code that most students should be able to pick up and explore.

5.1 VM

The *tiny x86* virtual machine consists of two logical parts, CPU and RAM. First, CPU and RAM implementation is described, followed by explanation of what does tick do. Finally, configuration of the VM is described.

5.1.1 CPU

CPU class combines many pieces together. It houses loaded program, register allocation table, instruction fetch and instruction decode pipeline stages, reservation station, branch predictor, branch predictions and physical registers. For ease of use it also houses RAM itself, even though it could be shared between multiple objects.
Instruction fetch and instruction decode pipeline stages are made of instruction pointer and next program counter. The next program counter is obtained by either branch predictor or incrementing current program counter and is used to set PC register for given instruction, more on this later. If fetched instruction is a jump instruction, checked by using C++ run-time type information, the branch predictor is invoked. The branch predictor takes current program counter and instruction reference. Currently, only naive branch predictor, always predicting, that the branch will be taken. This however has only effect if the target of jump instruction is an absolute address, otherwise PC increment is used.

5.1.1.1 Register allocation table

Register allocation table it self is mainly a wrapper for the mapping from logical registers to physical registers. Class copy constructors and destructors play huge role in keeping track of how many times a physical register is mapped to. Every time a RAT is copied, it reports the CPU what physical registers it maps to. In same fashion, destruction informs the CPU that it no longer maps to those registers. And for renaming/remapping of some logical register, first the old mapped register is unsubscribed and the newly mapped to is subscribed.

5.1.1.2 Reservation station

Reservation station handles its entries and available ALU count. Entries are stored in a linked list to keep the program order. Reference to hosting CPU is stored as well, used for obtaining operands and distributed to each entry for later use.
When instruction, along side next PC, is added to the RS, read RAT is created by copying the current CPU RAT and every product is extracted from instruction and processed. For register writes, renaming on CPU RAT happens, and for memory writes a write is registered. Also, each instruction changes the PC, so PC is renamed for each instruction, even if it does not influence PC by its execution [1]. Next a entry is created with previously copied read RAT, newly copied write RAT, registered memory write IDs and max memory write ID.
During operand fetching, a reservation station iterates through every entry in preparing state, checking if any operand of such entry is available to be supplied.
For execution, every executing entry's remaining execution ticks is decremented. If it hits zero, execute function is invoked on stored instruction and retiring state is set.
When retiring phase happens, the entry list begin is checked if ready for retirement. If so, retirement happens by invoking retire method on contained instruction. If the list begin is in non-retiring state, every subsequent entry waiting for retirement logs retirement stall. Reservation station entries store the instruction pointer, operands, read and write RATs and registered memory writes. Operands are copied from the instruction and modified during fetching. They are also used in instruction execution and retirement, more on this in section discussing instruction implementation. Entries provides methods for reading and writing registers, abstracting RAT translations from end user.

5.1.1.3 Operands

Operands are implemented using *std::variant* of all possible addressing modes. Base operand is integer or float value. Next, logical registers are just wrappers around register index. Iteratively, by combining defined operand, more complicated operands can be build, until all specified possible operands are built.
The operand class it self provides methods for checking and getting specific types. For easier manipulation, twin methods *requires* and *supply* are provided. Method *requires* returns a requirement object. Requirement is again internally represented as an *std::variant* and can be either register fetch or memory read. Once required value is obtained, *supply* with that value is

[1] This has to happen before copying the read RAT

called. This internally transforms the operand to a different type, based of current type. Operand is considered fetched once it a either float or integer value.
Operand also implements type getter. This type is used to create instruction signature as is represented as an enum.

5.1.1.4 Instructions

Thanks to the OOP design, individual instructions inherit from base instruction class, specifying method, which each instruction should implement. Such methods are:

- *length()* returning length of the instruction in bytes
- *type()* returning enum value of different instruction types (*MOV*, *ADD* etc.)
- *signatureOperands()* returning vector of instruction operands that should be displayed in instruction signature
- *operands()* returning vector of instruction operands - this contains operands, that are implicit for given instruction type, for example *JNZ* requires *FLAGS* register to determine if branch should be taken
- *validate()* validating its operands, for example *ADD PC 2* is not allowed by the ISA
- *produces()* returning vector of products/side-effects that this instruction execution creates
- *execute(ReservationStation::Entry)* that is called once execution of given instruction ends
- *retire(ReservationStation::Entry)* that is called during instruction retirement
-

By implementing these methods, instruction set can be extended quite easily for testing purposes by the students, where many of these methods can return dummy values.
Execute method uses entry to obtain and access fetched operands, set write address and value, and set registers. Similarly, in retirement method, entry provides a bridge to operands, memory writes and CPU.
Internally, instructions usually store the general operand object, so that restricting or extending available addressing modes for instructions boils down to whether a construction with that address mode is defined.

To eliminate repetition, some instruction subtypes, sharing common implementation, were defined, such as *JumpInstruction*, used to define branch instructions such as *JMP* and others. For even less code duplication, macros were used, especially for *BinaryArithInstruction*, *UnaryArithInstruction* and *ConditionalJumpInstruction* derived instructions, that differ only by different ALU function call or by a condition.

5.1.1.5 Speculation unrolling

When branch triggers speculation unrolling, the whole pipeline is cleared, pending writes are removed and write RAT from the triggering entry is set as new main CPU RAT. Speculation unrolling is also triggered when *DBG* or *BREAK* handlers are invoked, to put the CPU in predictable state and to guarantee any change would propagate correctly to next instructions.

5.1.2 RAM

RAM here servers not only as data storage, but also to emulate the latency real communication with RAM would have.
The data is stored in array of unsigned 64bit integers. During this class instantiation along side gate count a size is passed in parameter, allocating array of appropriate size and each field is set to zero. Each access is checked, so in case of out-ouf-bounds access an exception will be thrown.
Ongoing reads are stored in a map from address to a structure made of the value, that was stored at the time of request, and remaining ticks to be finished. By this, the read latency is simulated. Similarly, the ongoing writes are stored in a map from address to a structure, made of value to be written, write ID returned when write is requested to be able to track specific write, and remaining time to be finished.
If write is requested, an unique write ID is returned, write is added to ongoing writes and the value is already stored. This can be done, as the ISA specifies reading memory address, that is being written to, as undefined behavior. Using the write ID, the RAM can be asked, if given write has finished already, checking if ongoing write with such ID is stored.
Reading from an address returns *std::optional*. *std::nullopt* signals either RAM being busy, read started or read to that address is in progress. If read finished this tick, value of it is returned. When tick occurs, ongoing IO's remaining time is decremented. If remaining time goes to zero, it is marked for deletion at the beginning of next tick.
For debugging purposes, method for instantaneous setting and getting value to/from address, so that the change propagates for the next instructions.

5.1.3 Tick

CPU tick consists of multiple steps with intention of maximal throughput, but with behavior as close as possible to a read hardware. Unlike hardware, where during the tick multiple things interact with each other at once, this implementation has to sequentialize this process.
First, RAM tick is invoked, followed by executing and retiring phase in RS. Next up is operand fetching phase. Here, values resulting from execution of previous instruction can be used, hence this order. Entries, that were in the ready state (all operands are fetched) at the beginning of the tick, are now dispatched if possible - depending on ALU needs and availability.
After all pipeline stages happening in RS, instruction in instruction decode stage is forwarded to RS, setting created entry's state to preparing. If RS is full, stall happens, influencing instruction fetch stage as well, that would otherwise forwards its instruction to instruction decode stage and load a new instruction.
This order of steps is deliberately chosen to enforces minimum of 1 tick in each of the pipeline stages. The less apparent enforced tick is for operand fetching stage - when instruction is added to RS, operand fetching already happened, being marked as in operand fetching state even if no operand fetching is required. It might as well seem, that instruction retirement happens in the same tick as last instruction execution tick for given entry, but actually, when the entry is leaving preparing state, it is counted as first execution tick, so in the seemingly last instruction tick, only result forwarding happens. This also explains, why only entries, that were in ready state at the start of the tick, are processed, and not even those, that just entered this state by finished operand fetching. Note, that ready does not correspond to any pipeline stage, so if no stall happens, due to no ALU required or it being available, entry can be in this state only temporarily.

5.1.4 Configuration

As a part of the *tinyverse* project, basic configuration parsing from command line is implemented. This sort of a configuration does not allow fine configuration, but for basic VM parameters this is sufficient. For finer configuration we could use some JSON config file, or transparently set these values inside code, with easy way of changing these values.
For setting VM related parameters - register, ALU and reservation station entries counts, RAM size and gate counts are configurable through program arguments. Each of these has its own default value and switch for setting value. For example to set that CPU has 10 logical registers, "-registerCnt=10" would be added to the program arguments. All of these configuration strings can be found in this thesis attachment. These configurations are accessible during generating code for students to implement optimizations based on them using "Cpu::Config" singleton proving getters for each parameter specified by program arguments. Default values for unspecified parameters are set during initialization of mentioned singleton object.
For finer setting, a .cpp file implementing getters of such parameters is provided. By this method, you can specify RAM latency, instruction lengths, how long they execute and what operands do they take.
If such configuration file would be missing, linking of final executable would end with error, requesting supplying these methods. The envisioned use-case is that for different tasks a teacher could distribute such file with predetermined values. Such approach does require compilation of the distributed file and linking with the rest of the project, but this way no extra library for parsing configuration files is needed, leaving less discrepancies between student environments created by different ways of obtaining such library, and keeping the codebase concise without need of creating and maintaining own implementation or someone elses right inside the project.

5.2 Program and program building

Program consists of two separate segments - instructions and data. This models the Harvard architecture. Because instruction encoding end decoding is intentionally omitted, von Neumann architecture would be very hard to achieve.
Program builder class was created, to support program creation. It allows sequential program creation, data insertion and jump instruction patching. For easier operand definition, helper functions utilizing overloading are defined, allowing intuitive notation.
When instruction is added, a label, representing address in program, is returned, so that the instruction can be referenced as jump or call destination later in program. If forward jump is required, meaning that the destination address is unknown at the time of instruction addition, *Label::empty()* can be used as temporary value, that can be replaced later by calling *patch* method on program builder with the label of instruction, that requires patching, and the new target address.
Adding data will return a data label, representing address in memory, that will contain that data, once program is loaded. CPU stores data from data segment into the main memory, starting at address 0. Helper method for adding c-style string constant is provided, but it stores the string in simple, but highly space inefficient way. Students are welcome to implement their own extension for adding packed data.

```
using namespace tiny::t86;
ProgramBuilder pb;

DataLabel str =
        pb.addData("Hello world\n");

        pb.add(MOV{Reg(0), str});
        pb.add(MOV{Reg(1), 0});
Label loop =
        pb.add(MOV{Reg(2), Mem(Reg(0) + Reg(1))});
        pb.add(CMP{Reg(2), '\0'});
Label jumpToBody =
        pb.add(JNE(Label::empty()));
        pb.add(HALT{});
Label body =
        pb.add(PUTCHAR(Reg(2)));
        pb.add(INC{Reg(1)});
        pb.add(JMP{loop});

        pb.patch(jumpToBody, body);
```

Listing 5.1: Hello world program building example

5.3 Performance counters

To give effective feedback on compiled program, performance counting takes place. It consists of two phases - collection and processing.
Collection happens during the CPU execution, where individual components report state of the machine and causes of stalls. This creates very simplified snapshots of the CPU, used later for processing and displaying result information. Each pipeline stage reports its status to *StatsLogger* singleton object. For precise tracking inside reservation station, each entry is assigned its unique logging ID and provides helper methods for logging events using its assigned ID. Whenever operand requirement is not available, it is reported. For register fetching, a small trick here is deployed to provide as much information. If register fetch stall occurs, a fake dummy value is supplied to a separate copy of this operand and tested, if another register fetch is required, to log possible stall of this second register. This happens only is specific cases, limited by specified possible addressing modes. Better general implementation could be done by creating dependency graph for each operand, only having to check leave dependencies.
Processing happens after the CPU has halted. In this stage, lifetime of each instruction is reconstructed. All of these can be combined for average lifetime or grouped by instruction signatures for more detailed report. This can be outputted in text format, printed to console or stored in file for later in-depth analysis.

Chapter 6

Evaluation

This chapter describes validation compiler, showcases benefits of performance counting in benchmarking two different compiler techniques and discussed ongoing semester feedback.

6.1 Validation

Because students will use this virtual machine as target of their compiler backend as semestral work, similar compiler was implemented as validation of capability and usability of *tiny x86*.
A simple non-optimizing compiler targeting *tiny x86* was created. Same environment as current student have was used, including provided parser for *tinyC*. Every construct of input language was successfully translated, including loops, structs, arrays and functions and is a part of included implementation.

6.2 Performance counting

To showcase output of performance counter, simple *tinyC* program snippet was translated using two different translation techniques. First technique stores every variables on stack, loading it every time on access and storing on modification. Second uses register allocation principles to minimize memory accesses.

```
void foo() {
        int x = 0;
        for (int i = 1; i <= 10; ++i) {
                x += i;
        }
}
```

Listing 6.1: Sample *tinyC* code

```
        PUSH Bp // Save old function frame
        MOV Bp Sp // Create new function frame
        SUB Sp 2 // Allocating two local variables
        MOV [Bp - 1] 0 // x = 0

        // For loop
        MOV [Bp - 2] 1 // i = 1
loop:
        MOV R1 [Bp - 2] // Load i
        CMP R1 10 // Comparing i to 10
        JG ret // If i is greater, jump to end of the loop

        MOV R0 [Bp - 1] // Load x
        MOV R1 [Bp - 2] // Load i
        ADD R0 R1 // x += i
        MOV [Bp - 1] R0 // Store updated x
        MOV R1 [Bp - 2] // Load i
        INC R1 // ++i
        MOV [Bp - 2] R1 // Update i
        JMP loop
ret:
        MOV Sp Bp // Deallocate local variables
        POP Bp // Restore old function frame
        RET
```

Listing 6.2: Generated code n. 1

```
        PUSH Bp // Save old function frame
        MOV Bp Sp // Create new function frame
        MOV R0 0 // x = 0;

        // For loop
        MOV R1 1 // i = 1;
loop:
        CMP R1 10 // Compare i with 10
        JG ret // If i is greater, jump to end of the loop

        ADD R0 R1 // x += i
        INC R1 // ++i
        JMP loop
ret:
        POP Bp // Restore old function frame
        RET
```

Listing 6.3: Generated code n. 2

Selected code sample 6.1 is only for demonstrational purposes, it most likely has no real use and good optimizing compiler would optimize it just to *RET* as it has no observable side-effect and no return value, but that is not the point of this demonstration.
Listing 6.2 is a text representation of generated code[2] using the first mentioned technique, listing 6.3 using the second technique. For a skilled assembly programmer it might be obvious, that first technique requires much more memory accesses, but this is a code, that is expected for students to produce at some point and they might not be skilled enough to immediately recognize this or they might have some doubts regarding specific parts. This is where processed statistics from performance counting might help find and identify bottlenecks.

[2] with added comments

```
Total ticks: 551
Total instructions executed: 124
Throughput: 0.225045 instructions per tick
Average instruction latency: 4.44355 ticks
Global averages:
  Average instruction lifetime: 21.2823 ticks
        Average fetch stalls: 2.94355 ticks
        Average decode stalls: 3.20161 ticks
        Average operand fetching stalls: 8.12097 ticks
          Average register fetch stalls: 6.16129 ticks
          Average memory read stalls: 2.52419 ticks
        Average waiting for ALU: 0 ticks
        Average executing: 3 ticks
        Average waiting for retirement: 1.09677 ticks
        Average retirement: 1 ticks
```

Listing 6.4: Performace counter output for program 6.2

```
Total ticks: 207
Total instructions executed: 63
Throughput: 0.304348 instructions per tick
Average instruction latency: 3.28571 ticks
Global averages:
  Average instruction lifetime: 9.88889 ticks
    Average fetch stalls: 1.11111 ticks
    Average decode stalls: 1.28571 ticks
    Average operand fetching stalls: 1.57143 ticks
      Average register fetch stalls: 1.4127 ticks
      Average memory read stalls: 0.15873 ticks
    Average waiting for ALU: 0 ticks
    Average executing: 2.96825 ticks
    Average waiting for retirement: 0.142857 ticks
    Average retirement: 1 ticks
```

Listing 6.5: Performace counter output for program 6.3

Listing 6.4 shows performance counter output for the program 6.2 where RAM has read and write latency of 5 ticks. Listing 6.5 show output for program 6.3 on the same VM configuration. Both outputs can be compared, where the second technique shows better performance, as is expected. But even if we did not have the second program for comparison, student can see where stalls happens. Even more detailed output can be produced, displaying stats for individual instruction signatures.

6.3 Semester feedback

Due to still ongoing semester, no official feedback was collected, but from limited interaction with students, few observation were made. These observations are purely anecdotal and need confirmation, but show at least some sentiment.

Some students want to have a visual representation of what is happening inside the VM. This lead to them adding their own print statements, accessing internal state of the CPU. The design of the CPU does support debug interfaces, but they should be used in very specific conditions, that instructions like *DBG* or *BREAK* guarantee. Pointing them to using these instructions immediately solved this.

As a part of the compiling process, optimizations should be performed. Some

of these optimization can be done on already generated target code. In current design, this brings some problems, especially when trying to modify that program. New program has to be created, as current program builder does not support modifications of already added instructions, with the exception of patching jump instructions.

Conclusion

In this thesis, needs of the NI-GEN course were presented and based on them *tiny x86*, configurable architecture for educational purposes was designed.
Alongside *tiny x86* ISA, a virtual machine realizing this architecture was designed and implemented. For validation, simple non-optimizing compiler for *tinyC* programming language to this target was implemented as well.
At the time of publishing of this thesis, the *tiny x86* is being used in ongoing NI-GEN course as target for compiler backend created by students as their semestral work.

Future work

This section describes 3 main categories of possible future work - architectural extentions, supporting projects and potential ease of use changes.
Depending on needs of the NI-GEN course, more architectural designs can be added, to better demonstrate and exercise more advanced compilar techniques for diferent architectures without the need of full implementation of new target.
Similarly, VM design might be extended to accomodate more possible aspects for students to account to when creating their compilers. Memory caching is an area, that was not explored in depth in this thesis. Adding cache to the processors design and implementing such feature would unlock plethora of possible compiler techniques and optimizations that students could implement.
Many supporting sub-projects ehnancing the user experience could come out of this thesis. Components like GUI debugger would help students to intuitively analyze possible problems within their compiler generated code. It could also help extend the possible educational use-cases of this target - for example in courses like BI-SAP or BI-APS, that dive into processor design. Visual interactive statistics analyzer could help present output of the pro-

cessed performance counter in greater detail and readability.
Once the current NI-GEN course is finished, feedback from students should be collected and processed. Based on their feedback, improvements should be made, as usability of *tiny x86* is the most important goal of this thesis.

Bibliography

[1] Ampomah, E.; Mensah, E.; et al. Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages. *Communications on Applied Electronics*, volume 7, 10 2017: pp. 8–13, doi:10.5120/cae2017652685.

[2] Aho, A.; Lam, M.; et al. *Compilers. Principles, Techniques, and Tools (Second Edition)*. 01 2007.

[3] x86 continues to be the mainstream for server CPUs. (Accessed on 05.05.2021).

[4] `http://selfie.cs.uni-salzburg.at/`, (Accessed on 21.04.2021).

[5] Kirsch, C. M. Selfie and the Basics. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, New York, NY, USA: Association for Computing Machinery, 2017, ISBN 9781450355308, p. 198–213, doi:10.1145/3133850.3133857. Available from: `https://doi.org/10.1145/3133850.3133857`

[6] QEMU. `https://wiki.qemu.org/Main_Page`, (Accessed on 21.04.2021).

[7] Lattner, C.; Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[8] Clang: a C language family frontend for LLVM. (Accessed on 05.05.2021).

[9] LLVM Language Reference Manual. `https://llvm.org/docs/LangRef.html`, (Accessed on 05.05.2021).

[10] The LLVM Target-Independent Code Generator. https://llvm.org/docs/CodeGenerator.html, (Accessed on 05.05.2021).

[11] Intel. Microprocessor Quick Reference Guide. https://www.intel.com/pressroom/kits/quickreffam.htm, (Accessed on 06.05.2021).

[12] Intel. Introduction to Intel® Architecture. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf, (Accessed on 05.05.2021).

[13] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html, (Accessed on 06.05.2021).

[14] Sato, M. The Supercomputer "Fugaku" and Arm-SVE enabled A64FX processor for energy-efficiency and sustained application performance. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2020, pp. 1–5, doi:10.1109/ISPDC51135.2020.00009.

[15] MIPS32 Architecture – MIPS. https://www.mips.com/products/architectures/mips32-2/, (Accessed on 31.01.2021).

[16] Ramírez, A.; Larriba-Pey, J.-L.; et al. Branch Prediction Using Profile Data. 08 2001, pp. 386–393, doi:10.1007/3-540-44681-8_57.

[17] Jimenez, D.; Lin, C. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 197–206, doi:10.1109/HPCA.2001.903263.

[18] Tomasulo, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, volume 11, no. 1, 1967: pp. 25–33, doi:10.1147/rd.111.0025.

[19] Shen, J. P.; Lipasti, M. H. Modern Processor Design: Fundamentals of Superscalar Processors. 2002.

[20] Schlansker, M.; Rau, B. EPIC: Explicititly Parallel Instruction Computing. *IEEE Computer*, volume 33, 03 2000: pp. 37–45, doi:10.1109/2.820037.

Appendix A

Acronyms

CPU Central processing unit

ISA Instruction set architecture

VLIW Very long instruction word

VM Virtual machine

OS Operating system

CISC Complex instruction set computer

RISC Reduced instruction set computer

RS Reservation station

RAT Register allocation table

ALU Arithmetic logic unit

IO Input/output

RAM Random access memory

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `thesis` ..... the directory of LaTeX source codes of the thesis
  - `tinyverse` ..... the directory of tiny x86 implementation
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format

Appendix C

ISA documentation

This is documentation provided to the student at the begining of the course describing the ISA, its configuration and usage examples.

ISA

Flags

There are these flags

- OF - Overflow
- SF - Sign, copy of highest bit of result
- ZF - Zero, set if result is zero
- CF - Carry

Special Registers

- PC - Program counter
- FLAGS - Flags register
- SP - Stack pointer
- BP - Base pointer

Legend

- `R{X}` indicates any register, `X` is used to distinguish multiple registers
- `i{X}` indicates integer constant, `X` is used to distinguish multiple constants
- `F{X}` indicates any float register, `X` is used to distinguis multiple float registers
- `f{X}` indicates float constant, `X` is used to distinguish multiple constants
- `[{X}]` indicates access to memory, `X` indicates the address

Instructions details

General

Instruction	Operands	Description	Length (B)	Cycle time
MOV	`R1 , R2`	`R1 = R2`		
	`R1 , i`	`R1 = i`		
	`R1 , [i]`	`R1 = [1]`		
	`R1 , [R2]`	`R1 = [R2]`		

Instruction	Operands	Description	Length (B)	Cycle time
	`R1, [R2 + i]`	`R1 = [R2 + i]`		
	`R1, [R2 * i]`	`R1 = [R2 * i]`		
	`R1, [R2 + R3]`	`R1 = [R2 + R3]`		
	`R1, [R2 + R3 * i]`	`R1 = [R2 + R3 * i]`		
	`R1, [R2 + i + R3]`	`R1 = [R2 + i + R3]`		
	`R1, [R2 + i1 + R3 * i2]`	`R1 = [R2 + i1 + R3 * i2]`		
	`R1, F1`	`R1 = F1` (bit copy, use `NRW` to safely convert float to int)		
	`F1, f`	`F1 = f`		
	`F1, F2`	`F1 = F2`		
	`F1, R1`	`F1 = R1` (bit copy, use `EXT` to safely convert int to float)		
	`F1, [i]`	`F1 = [i]` (bit copy)		
	`F1, [R1]`	`F1 = [R1]` (bit copy)		
	`[i], R1`	`[i] = R1`		
	`[i], F1`	`[i] = F1` (bit copy)		
	`[i1], i2`	`[i1] = i2`		
	`[R1], R2`	`[R1] = R2`		
	`[R1], F1`	`[R1] = F1` (bit copy)		
	`[R1], i`	`[R1] = i`		
	`[R1 + i], R2`	`[R1 + i] = R2`		
	`[R1 + i], F1`	`[R1 + i] = F1` (bit copy)		

Instruction	Operands	Description	Length (B)	Cycle time
	`[R1 + i1], i2`	`[R1 + i1] = i2`		
	`[R1 * i], R2`	`[R1 * i] = R2`		
	`[R1 * i], F1`	`[R1 * i] = F1` (bit copy)		
	`[R1 * i1], i2`	`[R1 * i1] = i2`		
	`[R1 + R2], i`	`[R1 + R2] = i`		
	`[R1 + R2], R3`	`[R1 + R2] = R3`		
	`[R1 + R2], F1`	`[R1 + R2] = F1` (bit copy)		
	`[R1 + R2 * i1], i2`	`[R1 + R2 * i1] = i2`		
	`[R1 + R2 * i], R3`	`[R1 + R2 * i] = R3`		
	`[R1 + R2 * i], F1`	`[R1 + R2 * i] = F1` (bit copy)		
	`[R1 + i1 + R2], i2`	`[R1 + i1 + R2] = i2`		
	`[R1 + i + R2], R3`	`[R1 + i + R2] = R3`		
	`[R1 + i + R2], F1`	`[R1 + i + R2] = F1` (bit copy)		
	`[R1 + i1 + R2 * i2], R3`	`[R1 + i1 + R2 * i2] = R3`		
	`[R1 + i1 + R2 * i2], F1`	`[R1 + i1 + R2 * i2] = F1` (bit copy)		
	`[R1 + i1 + R2 * i2], i3`	`[R1 + i1 + R2 * i2] = i3`		

NOP | | Do nothing

Arithmetics

Instruction	Operands	Description	Length (B)	Cycle time
ADD	`R1 , R2`	`R1 += R2`		
	`R1 , i`	`R1 += i`		
	`R1 , R2 + i`	`R1 += R2 + i`		
	`R1 , [i]`	`R1 += [i]`		
	`R1 , [R2 + i]`	`R1 += [R2 + i]`		
SUB	`R1 , R2`	`R1 -= R2`		
	`R1 , i`	`R1 -= i`		
	`R1 , R2 + i`	`R1 -= R2 + i`		
	`R1 , [i]`	`R1 -= [i]`		
	`R1 , [R2 + i]`	`R1 -= [R2 + i]`		
INC	`R1`	`R1++`		
DEC	`R1`	`R1--`		
NEG	`R1`	`R1 = -R1`		
MUL	`R1 , R2`	`R1 *= R2`		
	`R1 , i`	`R1 *= i`		
	`R1 , R2 + i`	`R1 *= R2 + i`		
	`R1 , [i]`	`R1 *= [i]`		
	`R1 , [R2 + i]`	`R1 *= [R2 + i]`		
DIV	`R1 , R2`	`R1 /= R2`		
	`R1 , i`	`R1 /= i`		
	`R1 , R2 + i`	`R1 /= R2 + i`		
	`R1 , [i]`	`R1 /= [i]`		
	`R1 , [R2 + i]`	`R1 /= [R2 + i]`		

Instruction	Operands	Description	Length (B)	Cycle time
IMUL	`R1 , R2`	`R1 *= R2` , signed		
	`R1 , i`	`R1 *= i` , signed		
	`R1 , R2 + i`	`R1 *= R2 + i` , signed		
	`R1 , [i]`	`R1 *= [i]` , signed		
	`R1 , [R2 + i]`	`R1 *= [R2 + i]` , signed		
IDIV	`R1 , R2`	`R1 /= R2` , signed		
	`R1 , i`	`R1 /= i` , signed		
	`R1 , R2 + i`	`R1 /= R2 + i` , signed		
	`R1 , [i]`	`R1 /= [i]` , signed		
	`R1 , [R2 + i]`	`R1 /= [R2 + i]` , signed		

Float arithmetics

Instruction	Operands	Description	Length (B)	Cycle time
FADD	`F1 , F2`	`F1 += F2`		
	`F1 , f`	`F1 += f`		
FSUB	`F1 , F2`	`F1 -= F2`		
	`F1 , f`	`F1 -= f`		
FMUL	`F1 , F2`	`F1 *= F2`		
	`F1 , f`	`F1 *= f`		
FDIV	`F1 , F1`	`F1 /= F2`		
	`F1 , f`	`F1 /= f`		

Bit operations

Instruction	Operands	Description	Length (B)	Cycle time
AND	`R1 , R2`	`R1 &= R2`		

Instruction	Operands	Description	Length (B)	Cycle time
	`R1 , i`	`R1 &= i`		
	`R1 , R2 + i`	`R1 &= R2 + i`		
	`R1 , [i]`	`R1 &= [i]`		
	`R1 , [R2 + i]`	`R1 &= [R2 + i]`		
OR	`R1 , R2`	`R1 |= R2`		
	`R1 , i`	`R1 |= i`		
	`R1 , R2 + i`	`R1 |= R2 + i`		
	`R1 , [i]`	`R1 |= [i]`		
	`R1 , [R2 + i]`	`R1 |= [R2 + i]`		
XOR	`R1 , R2`	`R1 ^= R2`		
	`R1 , i`	`R1 ^= i`		
	`R1 , R2 + i`	`R1 ^= R2 + i`		
	`R1 , [i]`	`R1 ^= [i]`		
	`R1 , [R2 + i]`	`R1 ^= [R2 + i]`		
NOT	`R1`	`R1 = ~ R1`		
LSH	`R1 , R2`	`R1 <<= R2`		
	`R1 , i`	`R1 <<= i`		
	`R1 , R2 + i`	`R1 <<= R2 + i`		
	`R1 , [i]`	`R1 <<= [i]`		
	`R1 , [R2 + i]`	`R1 <<= [R2 + i]`		
RSH	`R1 , R2`	`R1 >>= R2`		
	`R1 , i`	`R1 >>= i`		
	`R1 , R2 + i`	`R1 >>= R2 + i`		
	`R1 , [i]`	`R1 >>= [i]`		
	`R1 , [R2 + i]`	`R1 >>= [R2 + i]`		

Compare

Instruction	Operands	Description	Detail	Length (B)	Cycle time
CMP	`R1, R2`	Compare `R1` with `R2`	Sets flags the same way `SUB` would		
	`R1, i`	Compare `R1` with `i`			
	`R1, [i]`	Compare `R1` with `[i]`			
	`R1, [R2]`	Compare `R1` with `[R2]`			
	`R1, [R2 + i]`	Compare `R1` with `[R2 + i]`			
FCMP	`F1, F2`	Compare `F1` with `F2`	Sets flags the same way `FSUB` would		
	`F1, f`	Compare `F1` with `f`			

Jump

Instruction	Operands	Description	Detailed condition	Length (B)	Cycle time
JMP	`R1`	Jump to value of `R1`			
	`i`	Jump to `i`			
LOOP	`R1, R2`	Jump to value of `R2` if `R1` != 0 and decrement `R1`			
	`R1, i`	Jump to `i` if `R1` != 0 and decrement `R1`			
JZ	`R1`	Jump to value of `R1` if zero	`ZF == 1`		
	`i`	Jump to `i` if zero			

Instruction	Operands	Description	Detailed condition	Length (B)	Cycle time
	`[i]`	Jump to value of `[i]` if zero			
	`[R1]`	Jump to value of `[R1]` if zero			
	`[R1 + i]`	Jump to value of `[R1 + i]` if zero			
JNZ	`R1`	Jump to value of `R1` if not zero	`ZF == 0`		
	`i`	Jump to `i` if not zero			
	`[i]`	Jump to value of `[i]` if not zero			
	`[R1]`	Jump to value of `[R1]` if not zero			
	`[R1 + i]`	Jump to value of `[R1 + i]` if not zero			
JE	`R1`	Jump to value of `R1` if equal	`ZF == 1`		
	`i`	Jump to `i` if equal			
	`[i]`	Jump to value of `[i]` if equal			
	`[R1]`	Jump to value of `[R1]` if equal			
	`[R1 + i]`	Jump to value of `[R1 + i]` if equal			
JNE	`R1`	Jump to value of `R1` if not equal	`ZF != 1`		
	`i`	Jump to `i` if not equal			
	`[i]`	Jump to value of `[i]` if not equal			

Instruction	Operands	Description	Detailed condition	Length (B)	Cycle time
	`[R1]`	Jump to value of `[R1]` if not equal			
	`[R1 + i]`	Jump to value of `[R1 + i]` if not equal			
JG	`R1`	Jump to value of `R1` if greater	`ZF == 0 && SF == OF`		
	`i`	Jump to `i` if greater			
	`[i]`	Jump to value of `[i]` if greater			
	`[R1]`	Jump to value of `[R1]` if greater			
	`[R1 + i]`	Jump to value of `[R1 + i]` if greater			
JGE	`R1`	Jump to value of `R1` if greater or equal	`SF == OF`		
	`i`	Jump to `i` if greater or equal			
	`[i]`	Jump to value of `[i]` if greater or equal			
	`[R1]`	Jump to value of `[R1]` if greater or equal			
	`[R1 + i]`	Jump to value of `[R1 + i]` if greater or equal			
JL	`R1`	Jump to value of `R1` if less	`SF != OF`		
	`i`	Jump to `i` if less			
	`[i]`	Jump to value of `[i]` if less			

Instruction	Operands	Description	Detailed condition	Length (B)	Cycle time
	`[R1]`	Jump to value of `[R1]` if less			
	`[R1 + i]`	Jump to value of `[R1 + i]` if less			
JLE	`R1`	Jump to value of `R1` if less or equal	`ZF == 1 && SF != OF`		
	`i`	Jump to `i` if less or equal			
	`[i]`	Jump to value of `[i]` if less or equal			
	`[R1]`	Jump to value of `[R1]` if less or equal			
	`[R1 + i]`	Jump to value of `[R1 + i]` if less or equal			
JA	`R1`	Jump to value of `R1` if above	`CF == 0 && ZF == 0`		
	`i`	Jump to `i` if above			
	`[i]`	Jump to value of `[i]` if above			
	`[R1]`	Jump to value of `[R1]` if above			
	`[R1 + i]`	Jump to value of `[R1 + i]` if above			
JAE	`R1`	Jump to value of `R1` if above or equal	`CF == 0`		
	`i`	Jump to `i` if above or equal			

Instruction	Operands	Description	Detailed condition	Length (B)	Cycle time
	`[i]`	Jump to value of `[i]` if above or equal			
	`[R1]`	Jump to value of `[R1]` if above or equal			
	`[R1 + i]`	Jump to value of `[R1 + i]` if above or equal			
JB	`R1`	Jump to value of `R1` if below	`CF == 1`		
	`i`	Jump to `i` if below			
	`[i]`	Jump to value of `[i]` if below			
	`[R1]`	Jump to value of `[R1]` if below			
	`[R1 + i]`	Jump to value of `[R1 + i]` if below			
JBE	`R1`	Jump to value of `R1` if below or equal	`CF == 1 || ZF == 1`		
	`i`	Jump to `i` if below or equal			
	`[i]`	Jump to value of `[i]` if below or equal			
	`[R1]`	Jump to value of `[R1]` if below or equal			
	`[R1 + i]`	Jump to value of `[R1 + i]` if below or equal			
JO	`R1`	Jump to value of `R1` if overflow	`OF == 1`		
	`i`	Jump to `i` if overflow			

Instruction	Operands	Description	Detailed condition	Length (B)	Cycle time
	`[i]`	Jump to value of `[i]` if overflow			
	`[R1]`	Jump to value of `[R1]` if overflow			
	`[R1 + i]`	Jump to value of `[R1 + i]` if overflow			
JNO	`R1`	Jump to value of `R1` if not everflow	`OF == 0`		
	`i`	Jump to `i` if not overflow			
	`[i]`	Jump to value of `[i]` if not overflow			
	`[R1]`	Jump to value of `[R1]` if not overflow			
	`[R1 + i]`	Jump to value of `[R1 + i]` if not overflow			
JS	`R1`	Jump to value of `R1` if sign	`SF == 1`		
	`i`	Jump to `i` if sign			
	`[i]`	Jump to value of `[i]` if sign			
	`[R1]`	Jump to value of `[R1]` if sign			
	`[R1 + i]`	Jump to value of `[R1 + i]` if sign			
JNS	`R1`	Jump to value of `R1` if not sign	`SF == 0`		
	`i`	Jump to `i` if not sign			

Instruction	Operands	Description	Detailed condition	Length (B)	Cycle time
	`[i]`	Jump to value of `[i]` if not sign			
	`[R1]`	Jump to value of `[R1]` if not sign			
	`[R1 + i]`	Jump to value of `[R1 + i]` if not sign			

Call

Instruction	Operands	Description	Length (B)	Cycle time
CALL	`R1`	Pushes current `PC` and jumps to value of `R1`		
	`i`	Pushes current PC and jumps to `i`		
RET		Jumps to top of stack and pops stack		

Stack

Instruction	Operands	Description	Length (B)	Cycle time
PUSH	`R1`	Pushes `R1` to stack		
	`i`	Pushes `i` to stack		
FPUSH	`F1`	Pushes `F1` to stack		
	`f`	Pushes `f` to stack		
POP	`R1`	Stores top of stack to `R1` and pops stack		
FPOP	`F1`	Stores top of stack to `F1` and pops stack		

I/O

Instruction	Operands	Description	Length (B)	Cycle time
PUTCHAR	`R1`	Prints `R1` as ASCII		
GETCHAR	`R1`	Loads char as ASCII from input to `R1`		

Float manipulation

Instruction	Operands	Description	Length (B)	Cycle time
EXT	`F1, R1`	extends value of `R1` to double and stores it into `F1`		
NRW	`R1, F1`	narrows value of `F1` to int and stores it into `R1`		

Other

Instruction	Operands	Description
DBG	debug function	executes debug function
BREAK		executes handle function
HALT		halts the CPU

VM

Note: All code examples expect you to use namespace `tiny::t86`, it is not enforced on you, it is omitted for better readability.

Configuration

You can configure your VM by adding arguments to the executed program
To set register count, use `-registerCnt=X` - default is 10 (you can use large number of registers to begin with).
To set float register count, use `-floatRegisterCnt=X` - default is 5.
To set number of ALUs, use `-aluCnt=X` - default is 1.
To set number of reservation station entries, use `-reservationStationEntriesCnt=X` - default is 2.
To set RAM size, use `-ram=X` - default is 1024 64bit values (so total size will be 8*X bytes).
To set RAM gate count, use `-ramGates=X` - default is 4.

Note: You can check config from like in this example:

```
Cpu::Config::instance().registerCnt();
```

Creating program

```
ProgramBuilder pb;
pb.add(MOV{Reg(0), 42});
pb.add(MOV{Mem(Reg(0) + 27), 23});
pb.add(HALT{});

return pb.program();
```

Note: Do not forget to add `HALT`, otherwise your program will run forever executing only `NOP`s.

Running program example

```
StatsLogger::instance().reset();
Cpu cpu;

cpu.start(std::move(program));
while (!cpu.halted()) {
    cpu.tick();
}
StatsLogger::instance().processBasicStats(std::cerr);
```

Note: For more detailed stats you can add:

```
StatsLogger::instance().processDetailedStats(std::cerr);
```

Patching labels

```
ProgramBuilder pb;
Label jumpToBody = pb.add(JMP{Label::empty()});
...
Label body = pb.add(MOV{Reg(0), 0});
...
pb.patch(jumpToBody, body);
```

Adding data

```
DataLabel str = pb.addData("Hello world\n");
pb.add(MOV{Reg(0), str});
```

Data will be stored starting on address 0 and further. **Note**: This string storing is very wasteful, you can create your own packed data (I am sure you will be rewarded extra points).

Accessing CPU registers

```
cpu.getRegister(Reg(0));
```

Accessing CPU memory

```
cpu.getMemory(Mem(0))
```

Note: Memory is addresable by 8bytes (64bit values)

Debug and handle function

Debug and handle functions have to have this function signature

```
void fn(Cpu&)
```

You can hook one handle function (executed on `BREAK`) by

```
cpu.connectBreakHandler(&fn);
```

or using c++11 lambdas.
Debug example:

```
pb.add(DBG{&fn});
```

Note that DBG will be added only if your `ProgramBuilder` was not given `true` argument indicating release environment.

Other notes

There are some example is `tests/targets/tiny86/programs.cpp` .
If you encounter any bug, please don't hesitate to report it.