### 1. Introduksjon

#### 1.1 Oppgavetekst

* Ledestjernen gjennom hele oppgaveløsningen: «Oppgaven går ut på å lage et system som ...»

#### 1. 2 Bakgrunn

* Sammenhengen utstyret/programmet/systemet skal benyttes – motivasjon
* Alt som studenten ikke har gjort selv
* Teori

### 2. Konstruksjon (*alt.*Materialer og metoder)

* Alt studenten(e) har gjort selv
* En redegjørelse for konstruksjon/programmering/løsningsmetode nøyaktig som den foreligger
* Skjema, kretskort
* Blokkdiagram
* Flytdiagram
* Forklaring av program på overordnet nivå
* Beskrivelse av eksperimentelle oppsett (*uten* resultater)

### 3. Resultat

* Objektive, målbare resultater
* Ytelse
* Presisjon
* Feilrater
* Kompatibilitet
* s-1 (alt som kan måles pr sekund)

### 4. Diskusjon

* Med utgangspunkt i oppgaveteksten (1.1), sammenlign med resultatene (3)
* Mulige alternative løsninger (som ikke ble valgt)
* Forslag til forbedringer
* Konklusjon (hvis den er mer enn et avsnitt, kan den bli kap 5)

### Vedlegg

* Referanseliste
* Økonomisk oversikt (prototyp/volumproduksjon)
* Datablad (hvis disse ikke finnes lett tilgjengelig på nett)
* Programlisting (hele prosjektstrukturen som .7z)
* Prosjektplan

#### 1.1 Oppgavetekst

This task is about creating a simulation of a rudimentary computer through software, for the purpose of teaching new students about some operating system aspects such as process scheduling and memory management. To further improve the ease of learning the program also needs a way to visualize these aspects in the form of a graphical user interface. The program simulates a CPU’s fetch-decode-execute cycle while integrating the said OS-level features process scheduling and memory management.

This will hopefully be used in the Operating Systems and System Programming course, to add to the learning experience. As currently in the course there is no way of visualizing the inner workings of how the system behaves. There will be some simplifications done, where deemed necessary. It would also be optimal if the students are able to code some modules to be used in the program, such as a scheduling policies, as part of some assignment. To fulfill this requirement the program needs to be somewhat modular and also be coded in GO.

///////////////

The aspects done:

Memory

MMU (Paging, freelist)

CPU fetch-Decode-execute. (Instructions and such)

OS

Processes

Fyne (GUI)

Loggin(SystemState)

Scheduling

/////////////

1. Konstruksjon

The Program is organized into two main parts, the OS and the dashboard. The OS hosts the rest of the underlying logic, like the CPU, Memory, Scheduler, MMU, Process Controller, and the Process table.These are all separate structs, which the OS holds a pointer for, inside the main OS struct. While the dashboard, is the logic required to make the GUI, which both show and allow some user inputs to operate on some data. Like creating processes and choosing which scheduler to use.

CPU Fetch-Decode-Execute cycle:

The basis of this simulation is CPU instructions, which are basic operations, which is done in the Execute stage by the CPU. An example of this is the Store instruction, which stores a value into the memory. A combination of these are combined into what is called a process, and this is what is being scheduled by a scheduler more info on this later.

Scheduling:

The scheduler is responsible for scheduling processes to be run on the CPU. When there are more than one process are ready to run on the CPU, the scheduler has to decide which takes priority. The simplest method is what is called FIFO(First in, First out), which is a queue system, where the first in line will be run next. There is a host of scheduling policies with its own benefits, drawbacks and problems, which will be touched upon later. First comes an explanation over how the scheduler is made. Inside the OS struct there is a scheduler interface. This is how modulaity of policies is done. Where you can input any scheduling policy inside the OS struct, as long as it fulfills the required interface requirements.

These are as follows:

AddProcess: This function adds a process to the ready queue(s) (Depends on policy).

GetNextProcess: This function returns the next process to be run based on the policy implemented.

GetRunningProcess: This function returns the current running process.

As long as the scheduling policy implements these, you can swap between any policy.

------------------------------

This project simulates a simplified CPU with an MMU, OS, memory, and peripherals to model the fetch, decode and execute cycle of the CPU.

1. CPU (Central Processing Unit)

- Fetches instructions from memory.

- Decodes the instruction

- Executes the instruction fetched from memory.

- Communicates with the MMU for memory access.

- Supports basic arithmetic and logical operations.

- Contains registers for temporary data storage.

- Program Counter, which points to the next instruction, in the code section in memory.

- Accumulator, which the logical operations work on.

- Instruction Register, holds the instruction in use.

- Memory Address Register, which holds the address for the memory.

- Memory Data Register, which hold either the instruction or data from memory.

The CPU has a few registers, used to store temporary data used to execute an instruction.

First the Program Counter (PC), which is used to point towards the next instruction in memory. The value stored is the virtual address not the physical address, so it needs to get translated by the MMU. This translated physical address is then stored in the Memory Address Register (MAR). During the fetching stage, the CPU looks at the Program counter, then asks the MMU to translate this virtual address to a physical one. The resulting physical address is then stored at the MAR. After this steps the CPU retrieves the instruction stored at the address inside the MAR. This retrieved instruction is temporary stored at the Memory Data Register (MDR). This register holds the retrieved data from memory, as such it doesn’t need to hold an instruction. If it is an instruction however, the next step is to store this to a register called Instruction register (IR), which holds the current running instruction, used for the other stages. After this is done, the program counter is increased by two, as as instruction uses two memory locations. One for the opcode/optype and the other is for the operand. These three values gets put into a instruction struct, and this struct is what is stored inside the IR. THis ends the fetching stage, and we proceed to the decoding stage.

At the decoding stage the CPU interprets the instruction. In my implementation it looks at the optype. This optype is a true or false indicating whether it needs to access the memory. If the optype is 0 (false), it interprets the operand as an actual value to be used at a later step. This value gets stored in the MDR, and the step is done. If the optype is 1, it skips this step. The optype being 1, indicates that the operand is a virtual address to be used to access memory, and the value stored at this memory location is what is stored inside the MDR to be used for the later operation, e.g ADD. However this memory accessing is also done in the execution stage, so the decoding stage is only to decipher the optype.

The next and last stage is the execution stage, where the instruction is executed. First it checks if it needs to access the memory, from the decoding stage. If it does, it does so first. Then the CPU proceeds to execute the opcode stored in the IR. These opcodes are mapped to operation functions stored inside the CPU. So it looks for the corresponding function for the opcode, and runs the function. This concludes the fetch-decode-execute cycle.

There are however one register not touched upon, which is used for the specific operation, namely the Accumulator (AC). It is this value that gets worked on by the operations. Lets look at the ADD operation. As the name suggests it adds a value on top of another, and the other being the accumulator. So it takes the current value in the accumulator, and adds the new value from the MDR. You can’t specify which value to add to, everything goes through the accumulator.

The same applies to the STORE operation, which stores the current value in the accumulator and stores it into a memory address.

Operations:

ADD:

Adds the specified value to the accumulator.

OldAccumulator + Value = NewAccumulator

SUB:

Subtract the specified value from the accumulator.

OldAccumulator - Value = NewAccumulator

STORE :

Stores the current value in the accumulator, to the specified memory address.

LOAD

Retrieves the value from the specified memory address, and sets the accumulator to it.

JUMP:

Sets the Program Counter to the specified value.

Note: The specified value can be directly from the operand, or from a stored memory address. Remember Optype.

2. MMU (Memory Management Unit)

- Translates virtual addresses to physical addresses.

- Reads from memory.

- Writes to memory.

1. OS (Operating System)

- The entity of that govern the other entities, like cpu, memory, scheduler, process controller.

- Initialize the required components:

- CPU

- Memory

- ProcessTable

- CPU Controller

- Scheduler

- FreeList

- Start/Stop/Pause/Resumes the simulation.

- Schedules processes.

1. Memory (RAM)

- Stores code, stack and heap, of various processes.

- Accessible by the CPU via the MMU.

5. Scheduler

- Manages the process queue, for which to run.

6. Process Controller

- Create a new process.

- Control the allocation of memory space.

- Store instructions

-

MMU functions:

func (mmu \*MMU) TranslateAddress(virtualAddr uint32) (int, error) {}

Takes in a virtual address, and returns the physical address given after a translation. This physical address is used to access the physical memory.

func (mmu \*MMU) Read(physicalAddr uint32) (int, error) {}

Takes in a physical address, reads from memory at that address, and returns the stored value at this address.

func (mmu \*MMU) Write(physicalAddr uint32, value uint32) error {}

Takes in a physical address and a uint32 value, goes to that memory address, and replaces the currently stored value, with the new one.