Technical System Specification

github.com/404dcd

2022-11-12

Contents

Chapter 1

Instruction Format and Codings

Instructions have a variable length but are always byte-aligned: padded with zeros at the end until it is $8 \times n$ bits long. The general structure of an instruction is as follows:

`[opc] [src type] [dst type] [src] [dst]`

The `src` fields (including type) are omitted for single-operand instructions, and both `src` and `dst` including types are omitted for instructions without operands.

1.1 Operand types

Code	Meaning	Bit layout
0000	`r`	4
0001	`immX`	8 / 16 / 32
0010	`uimm8`	8
0011	`[uimm32]`	32
0100	`[r]`	4
0101	`[r + uimm8]`	4 + 8
0110	`[r - uimm8]`	4 + 8
0111	`[r + uimm32]`	4 + 32
1000	`[r + r]`	4 + 4
1001	`[r + r*2]`	4 + 4
1010	`[r + r*4]`	4 + 4
1011	`[r + r*8]`	4 + 4
1100	`[uimm32 + r + r]`	32 + 4 + 4
1101	`[uimm32 + r + r*2]`	32 + 4 + 4
1110	`[uimm32 + r + r*4]`	32 + 4 + 4
1111	`[uimm32 + r + r*8]`	32 + 4 + 4

Square brackets (`[...]`) represent the treatment of `...` as a memory address that is then implicitly dereferenced. `r` represents a register identifier, as specified in the table below. `uimm` is an unsigned immediate value that is zero-extended to match wherever needed, and `immX` represents an immediate value that must be specified in the size of the operands. Thus, `[dst]` and `[src]` are more accurately referred to as *structs*, the field(s) of which are described by the corresponding operand type.

Unless otherwise specified, the following conditions must be met:

- A maximum of one operand is a memory reference
- The source operand may use any mode or any register
- The destination operand must not be an immediate, and must not be the IP register (it must be writable)

1.2 Registers

Code	Mnemonic	Description
0000	`ZR`	hardwired to 0
0001	`AX`	general purpose A
0010	`BX`	general purpose B
0011	`CX`	general purpose C
0100	`DX`	general purpose D
0101	`EX`	general purpose E
0110	`FX`	general purpose F
0111	`GX`	general purpose G
1000	`HX`	general purpose H
1001	`IX`	general purpose I
1010	`JX`	general purpose J
1011	`KX`	general purpose K
1100	`IM`	implicit register
1101	`SP`	stack pointer
1110	`BP`	base pointer
1111	`IP`	instruction pointer

All registers are 32 bits (the word size). The instruction pointer is read-only when it comes to regular register operations, and it may only be changed with control flow instructions or saving/restoring state. There are further special/control registers:

- PDBR - page directory base register. Stores the physical memory address of the first entry in the page directory for the currently running process.
- IVTR - interrupt vector table register. Stores the physical memory address (regardless of VMF) of the first entry in the interrupt vector table.
- FLGR - contains a bit field of the current flag status of the processor.

1.3 Opcodes

Code	Mnemonic	Description
0x00	`--`	Opcode extension - read next byte
0x01	`ADD`	Integer addition
0x02	`SUB`	Integer subtraction
0x03	`DSUB`	Result-discarding `SUB`
0x04	`INC`	Increment register
0x05	`DEC`	Decrement register

0x06	`AND`	Bitwise and
0x07	`DAND`	Result-discarding `AND`
0x08	`ORR`	Bitwise or
0x09	`XOR`	Bitwise exclusive or
0x0a	`NOT`	Bitwise inversion
0x0b	`NEG`	Two's complement negation
0x0c	`MUL`	Integer multiplication
0x0d	`SML`	Signed integer multiplication
0x0e	`DIV`	Integer division
0x0f	`SDV`	Signed integer division
0x10	`CPY`	Copy data
0x11	`SWP`	Swap data
0x12	`ASR`	Arithmetic shift right
0x13	`BSR`	Bitwise (logical) shift right
0x14	`BSL`	Bitwise (logical) shift left
0x15	`CSR`	Circular shift right (rotate)
0x16	`CSL`	Circular shift left (rotate)
0x17	`SNX`	Sign extend
0x18	`ZRX`	Zero extend
0x19	`LMA`	Load effective address
0x1a	`PUSH`	Append register to stack
0x1b	`POP`	Read stack off into register
0x1c	`PUSHR`	Push `AX - FX`
0x1d	`POPR`	Pop `FX - AX`
0x1e	`CPFLGR`	Copy the FLGR special register
0x1f	`CPIVTR`	Copy the IVTR special register
0x20	`WRIVTR`	Write the IVTR special register
0x21	`WRPDBR`	Write the PDBR special register
0x22	`SETIEF`	Enable interrupts
0x23	`CLRIEF`	Disable interrupts
0x24	`SETVMF`	Enable virtual memory
0x25	`CLRVMF`	Disable virtual memory
0x26	`JUMP`	Unconditional jump
0x27	`JAOE`	Jump if above or equal
0x28	`JABV`	Jump if above
0x29	`JBOE`	Jump if below or equal
0x2a	`JBEL`	Jump if below
0x2b	`JGOE`	Jump if greater or equal
0x2c	`JGRA`	Jump if greater
0x2d	`JLOE`	Jump if less or equal
0x2e	`JLES`	Jump if less
0x2f	`JSMM`	Jump if sign mismatch
0x30	`JNSM`	Jump if no sign mismatch
0x31	`JZRO`	Jump if zero
0x32	`JNZR`	Jump if not zero
0x33	`JPOS`	Jump if no sign
0x34	`JNEG`	Jump if sign
0x35	`CALL`	Call a function

0x36	`RET`	Return from a function
0x37	`INP`	Read data from port
0x38	`OUT`	Write data to port
0x39	`GENINT`	Generate a software interrupt
0x3a	`IRET`	Return from interrupt handler
0x3b	`NOP`	Explicitly do nothing
0x3c	`HLT`	Stop execution
0xfe	`.8`	Prefix - operand size is 8 bits
0xff	`.16`	Prefix - operand size is 16 bits

Opcode 0x00 is not currently in use, because the CPU has no need to use more than 255 opcodes. It is left for future compatibility for an extension set that all begin with 0x00, and so the whole opcode would be two bytes for this extension set.

Another case where the opcode is two bytes is when a prefix is used - either zero or one of the prefixes `.8` and `.16` may be used to precede an opcode. This changes the width of data the processor is operating on to 8 or 16 bits respectively. Concerning registers, generally this will cause the CPU to only consider the lowest 8 or 16 bits - reading from them only and writing back, in-place, without changing the upper bits (although there are exceptions: see `SNX` and `ZRX` in section 1.6)

1.4 Flags

The flags are primarily accessible through conditional jump instructions, but they are also wired into the FLGR register which can be read. For each of the following, the `number` indicates the power of 2 of the bit that they're located at (`0` as LSB etc)

- SMF `0` - sign mismatch, often called the Overflow flag on other architectures. Working on the assumption of using signed arithmetic: if subtracting a positive from a negative yields a positive number, or if subtracting a negative from a positive yields a negative number. More generally, SMF is set if the sign bit is not actually what the mathematical laws governing the operation would expect.
- COF `1` - carry out, indicates a carry out of the most significant bit. Working on the assumption of using unsigned arithmetic: if the result of an addition is too large to fit, or if a subtraction "bites off more than it can chew".
- ZRF `2` - zero, set if the result (value being copied into the destination) is equal to 0.
- NGF `3` - negative, set if the result had the sign bit (most significant bit) set.
- IEF `4` - interrupts enabled. The processor responds to interrupts if this flag is set (see section 2.1).
- VMF `5` - virtual memory mode enabled. The processor will translate all addresses using page tables if this flag is set (see section 2.2).

1.5 Detailed Behaviour - Arithmetic

ADD

Performs arithmetic addition of the source to the destination.
Available prefixes: `.8` and `.16`
Sets flags: SMF, COF, ZRF, NGF
Operands needed: `src` and `dst`

SUB

Performs arithmetic subtraction of the source from the destination.
Available prefixes: `.8` and `.16`
Sets flags: SMF, COF, ZRF, NGF
Operands needed: `src` and `dst`

DSUB

Performs arithmetic subtraction of the source from the destination, but preserves the destination by discarding the result. The destination has no restrictions and may be an immediate or the IP register.
Available prefixes: `.8` and `.16`
Sets flags: SMF, COF, ZRF, NGF
Operands needed: `src` and `dst`

INC

Adds the unsigned integer 1 to the destination. Preserves the COF flag.
Available prefixes: `.8` and `.16`
Sets flags: SMF, ZRF, NGF
Operands needed: `dst`

DEC

Subtracts the unsigned integer 1 from the destination. Preserves the COF flag.
Available prefixes: `.8` and `.16`
Sets flags: SMF, ZRF, NGF
Operands needed: `dst`

AND

Performs a bitwise logical AND (conjunction) of the source and the destination.
Available prefixes: `.8` and `.16`
Sets flags: ZRF, NGF
Operands needed: `src` and `dst`

DAND

Performs a bitwise logical AND (conjunction) of the source and the destination, but preserves the destination by discarding the result. The destination has no restrictions and may be an immediate or the IP register.

Available prefixes: `.8` and `.16`
Sets flags: ZRF, NGF
Operands needed: `src` and `dst`

ORR

Performs a bitwise logical OR (disjunction) of the source and the destination.

Available prefixes: `.8` and `.16`
Sets flags: ZRF, NGF
Operands needed: `src` and `dst`

XOR

Performs a bitwise logical exclusive OR (exclusive disjunction) of the source and the destination.

Available prefixes: `.8` and `.16`
Sets flags: ZRF, NGF
Operands needed: `src` and `dst`

NOT

Performs a bitwise logical negation (complement) of the destination.

Available prefixes: `.8` and `.16`
Sets flags: ZRF, NGF
Operands needed: `dst`

NEG

Calculates the arithmetic result of the negative of the destination in two's complement. Equivalent to subtraction of the destination from 0 (flags SMF and COF are set using the normal subtraction rules).

Available prefixes: `.8` and `.16`
Sets flags: SMF, COF, ZRF, NGF
Operands needed: `dst`

MUL

Performs unsigned multiplication of the source and the destination. The top half of the result is stored in the implicit register `IM`, using prefixes in the same fashion as the regular destination - if a prefix of `.8` is used, the lowest 8 bits of `IM` will be written and the lowest 8 bits of the destination also. COF is set if the top half of the result *is not* zero, ZRF is set if the bottom half of the result *is* zero.

Available prefixes: `.8` and `.16`
Sets flags: COF, ZRF

Operands needed: `src` and `dst`

SML

Performs signed multiplication of the source and the destination. The top half of the result is stored in the implicit register `IM`, using prefixes in the same fashion as the regular destination - if a prefix of `.8` is used, the lowest 8 bits of `IM` will be written and the lowest 8 bits of the destination also. COF is set if, when the bottom half of the result is sign-extended, it does not match the full result (i.e. the top half contains important data). ZRF is set if the bottom half of the result *is* zero.

Available prefixes: `.8` and `.16`
Sets flags: COF, ZRF
Operands needed: `src` and `dst`

DIV

Performs unsigned division of the destination by the source. The quotient (number of whole multiples) is stored in the destination, and the remainder is stored in the implicit register `IM`. Prefixes affect the writing of both registers in the same way as in multiplication. ZRF is set if the remainder (`IM`) is zero. Divide by zero exception is raised if the source is zero.

Available prefixes: `.8` and `.16`
Sets flags: ZRF
Operands needed: `src` and `dst`

SDV

Performs signed division of the destination by the source. The quotient is stored in the destination, and the remainder is stored in the implicit register `IM`. Prefixes affect the writing of both registers in the same way as in multiplication. The quotient and/or the remainder may be negative, such that arithmetically the $quotient \times source + remainder = destination$. ZRF is set if the remainder (`IM`) is zero. Divide by zero exception is raised if the source is zero. COF is set in the special case that a computation of the maximum negative value divided by negative 1 (the resulting positive number cannot fit) is attempted.

Available prefixes: `.8` and `.16`
Sets flags: COF, ZRF
Operands needed: `src` and `dst`

1.6 Detailed Behaviour - Control

CPY

Copies the source to the destination. Specially, both operands may be a memory reference.

Available prefixes: `.8` and `.16`
Sets flags: none
Operands needed: `src` and `dst`

SWP

Swaps the destination and the source.

Available prefixes: `.8` and `.16`
Sets flags: none
Operands needed: `src` and `dst`

ASR

Performs an arithmetic (sign-extending) shift to the right of the destination, with a shift count specified by the source. The shift count may be greater than or equal to the destination width, above which behaviour is defined in that the whole destination will be all ones or all zeros depending on the original sign (which gets duplicated into the MSB on every shift). The COF contains the most recently shifted-out bit.

Available prefixes: `.8` and `.16`
Sets flags: COF, ZRF, NGF
Operands needed: `src` and `dst`

BSR

Performs a bitwise (treating as unsigned) shift to the right of the destination, with a shift count specified by the source. The shift count may be greater than or equal to the destination width, above which behaviour is defined in that the whole destination will be all zeros. The COF contains the most recently shifted-out bit.

Available prefixes: `.8` and `.16`
Sets flags: COF, ZRF
Operands needed: `src` and `dst`

BSL

Performs a bitwise (treating as unsigned) shift to the left of the destination, with a shift count specified by the source. The shift count may be greater than or equal to the destination width, above which behaviour is defined in that the whole destination will be all zeros. The COF contains the most recently shifted-out bit.

Available prefixes: `.8` and `.16`
Sets flags: COF, ZRF
Operands needed: `src` and `dst`

CSR

Performs a circular shift to the right of the destination, with a shift count specified by the source. The shift count may be greater than or equal to the destination width, as it is taken modulo the destination width. Each shifted-out bit becomes the new MSB, and is also copied into the COF.

Available prefixes: `.8` and `.16`
Sets flags: COF, ZRF
Operands needed: `src` and `dst`

CSL

Performs a circular shift to the left of the destination, with a shift count specified by the source. The shift count may be greater than or equal to the destination width, as it is taken modulo the destination width. Each shifted-out bit becomes the new LSB, and is also copied into the COF.

Available prefixes: `.8` and `.16`
Sets flags: COF, ZRF
Operands needed: `src` and `dst`

SNX

Sign-extends a signed value, treated as the width of the prefix in the bottom bits of the destination register, into the whole 32-bit destination register. A prefix must be used with this instruction, and the destination format must be code 0000 - `r`.

Available prefixes: `.8` and `.16`
Sets flags: ZRF, NGF
Operands needed: `dst`

ZRX

Zero-extends an unsigned value, treated as the width of the prefix in the bottom bits of the destination register, into the whole 32-bit destination register. A prefix must be used with this instruction, and the destination format must be code 0000 - `r`.

Available prefixes: `.8` and `.16`
Sets flags: ZRF
Operands needed: `dst`

LMA

Loads the effective address of the source operand (which must be a memory reference, an operand code greater than or equal to 0011 - `[uimm32]`) into the destination (which must be code 0000 - `r`). Note that if a prefix is used the full-width operand must still be specified, and it will be truncated.

Available prefixes: `.8` and `.16`
Sets flags: none
Operands needed: `src` and `dst`

PUSH

Pushes the destination onto the stack. The destination format has no restrictions, it may be an immediate or the IP register. The width of the value being pushed (specified either by prefix or lack thereof) in bytes is subtracted from the stack pointer `SP` register, and the value then written from that new address towards larger addresses in memory, most significant byte first.

Available prefixes: `.8` and `.16`
Sets flags: none
Operands needed: `dst`

POP

Pops the top of the stack into the destination. The value on the stack starting at the address in the stack pointer `SP` is read into the destination, for as many bytes as the prefix (or lack thereof) specifies, towards larger addresses in memory, most significant byte first. Then the width of the value popped is added to the `SP` register.

Available prefixes: `.8` and `.16`
Sets flags: none
Operands needed: `dst`

PUSHR

Performs a full-width `PUSH` instruction four times, for destination registers `AX`, `BX`, `CX` and `DX`, `EX` and `FX` in that order.

Available prefixes: none
Sets flags: none
Operands needed: none

POPR

Performs a full-width `POP` instruction four times, for destination registers `FX`, `EX`, `DX`, `CX`, `BX` and `AX` in that order.

Available prefixes: none
Sets flags: none
Operands needed: none

CPFLGR

Copies the FLGR special register to the destination.

Available prefixes: none
Sets flags: none
Operands needed: `dst`

CPIVTR

Copies the IVTR special register to the destination.

Available prefixes: none
Sets flags: none
Operands needed: `dst`

WRIVTR

Writes the value from the destination (which has no restrictions, it may be an immediate or the IP register) into the IVTR special register.

Available prefixes: none
Sets flags: none
Operands needed: `dst`

WRPDBR

Writes the value from the destination (which has no restrictions, it may be an immediate or the IP register) into the PDBR special register.

Available prefixes: none
Sets flags: none
Operands needed: `dst`

SETIEF

Sets the IEF flag to 1, enabling the processor to respond to interrupts.

Available prefixes: none
Sets flags: IEF
Operands needed: none

CLRIEF

Sets the IEF flag to 0, disabling the processor from responding to interrupts.

Available prefixes: none
Sets flags: IEF
Operands needed: none

SETVMF

Sets the VMF flag to 1, enabling page translation of memory addresses and thus using the virtual memory mode.

Available prefixes: none
Sets flags: VMF
Operands needed: none

CLRVMF

Sets the VMF flag to 0, disabling page translation of memory addresses and thus using the physical memory mode.

Available prefixes: none
Sets flags: VMF
Operands needed: none

1.7 Detailed Behaviour - JXXX

The following J- instructions jump if a condition in the flags register is met. Their names are largely based on the flags generated by the `DSUB` (or regular subtraction) instruction, but can be used at any time. Only the destination operand is supplied, but it may not use the format codes 0000 through 0010 inclusive - it must be a memory reference. However, that reference is *not* dereferenced, instead the effective address is loaded into the instruction pointer and execution continues. This allows easy register-relative jumps, at the cost of not being able to jump to an address itself stored in memory in a single

instruction. This method of addressing is also employed for the `CALL` instruction. For all J- jump instructions:

Available prefixes: none
Sets flags: none
Operands needed: `dst`

JUMP

Performs an unconditional jump.

JAOE

Jumps if COF=0. If the previous flag-setting instruction was a subtraction, the jump is taken if the unsigned representation of the destination was larger than or equal to the unsigned representation of the source.

JABV

Jumps if COF=0 and ZRF=0. If the previous flag-setting instruction was a subtraction, the jump is taken if the unsigned representation of the destination was strictly larger than the unsigned representation of the source.

JBOE

Jumps if COF=1 or ZRF=1. If the previous flag-setting instruction was a subtraction, the jump is taken if the unsigned representation of the destination was smaller than or equal to the unsigned representation of the source.

JBEL

Jumps if COF=1. If the previous flag-setting instruction was a subtraction, the jump is taken if the unsigned representation of the destination was strictly smaller than the unsigned representation of the source.

JGOE

Jumps if SMF=COF. If the previous flag-setting instruction was a subtraction, the jump is taken if the signed representation of the destination was larger than or equal to the signed representation of the source.

JGRA

Jumps if SMF=COF and ZRF=0. If the previous flag-setting instruction was a subtraction, the jump is taken if the signed representation of the destination was strictly larger than the signed representation of the source.

JLOE

Jumps if SMF≠COF or ZRF=1. If the previous flag-setting instruction was a subtraction, the jump is taken if the signed representation of the destination was smaller than or equal to the signed representation of the source.

JLES

Jumps if SMF≠COF. If the previous flag-setting instruction was a subtraction, the jump is taken if the signed representation of the destination was strictly smaller than the signed representation of the source.

JSMM

Jumps if SMF=1. If the previous flag-setting instruction was a subtraction or an addition, the jump is taken if the maths on the signed representations of the operands caused a result too big or too small.

JNSM

Jumps if SMF=0. If the previous flag-setting instruction was a subtraction or an addition, the jump is taken if the maths on the signed representations of the operands produced a correct result.

JZRO

Jumps if ZRF=1. If the previous flag-setting instruction was a subtraction, the jump is taken if the destination was equal to the source.

JNZR

Jumps if ZRF=0. If the previous flag-setting instruction was a subtraction, the jump is taken if the destination was not equal to the source.

JPOS

Jumps if NGF=0. If the result is treated as a signed value, the jump is taken if it was a positive number.

JNEG

Jumps if NGF=1. If the result is treated as a signed value, the jump is taken if it was a negative number.

CALL

Performs the procedure for calling a function. The function's entry address is given by the destination operand, using the same addressing convention as `JXXX` jump instructions (loads effective address). During the instruction, a full-width `PUSH` is performed of the

new instruction pointer (for the next instruction after the call) onto the stack. Then an unconditional jump to the address given is executed.

Available prefixes: none
Sets flags: none
Operands needed: `dst`

RET

Performs the procedure for returning from a function. A full-width `POP` into the instruction pointer is executed, causing a jump back to the calling code.

Available prefixes: none
Sets flags: none
Operands needed: none

INP

Copies the data from a port (specified by the source operand which must be code 0010 - `uimm8`) to a register (the destination operand, which must be code 0000 - `r`). The port is modelled as a 32 bit register that can be changed by a hardware device at any time, and it should signal data is ready with an interrupt, and so prefixes just change how much of that register is read.

Available prefixes: `.8` and `.16`
Sets flags: none
Operands needed: `src` and `dst`

OUT

Copies the data from a register (specified by the destination operand which must be code 0000 - `r` and may be the IP) to a port (the source operand, which must be code 0010 - `uimm8`). The port is modelled as a 32 bit register that can be read by a hardware device at any time, and the CPU takes care of notifying the hardware device that new data is ready. A full 32-bit value must be written each time.

Available prefixes: none
Sets flags: none
Operands needed: `src` and `dst`

GENINT

Generates a software interrupt, the interrupt number being given by the destination operand which must be code 0010 - `uimm8`. A full-width `PUSH` is performed of the FLGR register, followed by the return address (pointing to the next instruction to be executed) onto the stack. Then an unconditional jump to the address given is executed. The interrupt number may not be one of those already specified in section 2.1, and may not be greater than 1023.

Available prefixes: none
Sets flags: none
Operands needed: `dst`

IRET

Returns from a software interrupt. A full-width `POP` into the instruction pointer is executed, and then the stack is popped again into the FLGR register. This causes a jump back to the code running before the interrupt occurred, and a full restoration of state given that the interrupt handler has preserved all other registers.

Available prefixes: none
Sets flags: all flags are restored
Operands needed: none

NOP

Explicitly does nothing. No operation.

Available prefixes: none
Sets flags: none
Operands needed: none

HLT

Pauses execution of the processor, which may only be started again by an interrupt. This instruction is only useful if interrupts are enabled (IEF is set).

Available prefixes: none
Sets flags: none
Operands needed: none

Chapter 2

The CPU

2.1 Interrupts

If the IEF flag is set, the CPU checks for interrupts at the end of each instruction cycle. If it finds there is one, it pushes the FLGR register on to the stack followed by the current IP (which is the returning address, for the next instruction). Just after it pushes the FLGR register, it sets the IEF flag to 0 to disable interrupts. Then, it uses the interrupt number (an integer between 0 and 1023 inclusive) to index into the interrupt vector table, a page of memory referenced by the physical address in the IVTR special register. Each entry in the IVT is 4 bytes - a memory address of the handler in the kernel to jump to.

It is the handler's responsibility to preserve all registers during its execution, unless explicitly stated otherwise. The handler may push onto the stack in the memory space of the process that was running, but it must be tidied up before the return. To return from the handler, the kernel can use `IRET` to pop the stack twice - the first into the IP register and then into the FLGR register to restore state seamlessly.

CPU exceptions (E.), hardware interrupts and software-generated interrupts are all treated as interrupts in this way. The following CPU exceptions and hardware interrupts are defined: and then the OS is free to pick any others to be its system calls.

Code	Source	Description
00	CPU E.	Divide by zero
01	CPU E.	Invalid opcode
02	CPU E.	Illegal instruction
03	CPU E.	Unpaged address
04	CPU E.	Null pointer dereference
05	CPU E.	Address beyond maximum
06	CPU E.	Unregistered interrupt
07 - 0f	–	Reserved
10	Keyboard	Key pressed
11	Keyboard	Metadata available
12	Disk	Data read into memory
13	Disk	Data written to disk
14	Disk	Metadata available
15	Memory	Metadata available

Illegal instruction is distinct to invalid opcode in that the former is raised when something is wrong with the operands of the instruction. Invalid opcode means that

the opcode itself is unrecognised to begin with. Unpaged address is raised when, while translating a virtual address to a physical address, the entry in the page directory or page table is 0. Exceptions differ from interrupts because the saved `IP` will point to the instruction that caused the exception - other than that they are the same.

If the IEF flag is not set: software interrupts will be ignored, hardware interrupts will be queued (the size of the queue depends on the implementation, in this emulator it is 128 items long) and exceptions will cause the CPU to halt. The IEF flag is cleared as soon as an interrupt/exception is detected, but it must be re-set manually by the handler routine or other code. The IEF flag is initialised to 0 at processor reset.

2.2 Hardware

CPU Ports

Code	Device
00	Memory controller
01	Serial out
02	Disk
03	Keyboard
04	Display screen

To write to or read from the ports of these devices, use the `OUT` and `INP` instructions respectively. Both of these operations involve a single 32-bit value at a time. Writing to a port will push the value to be sent onto a queue for the hardware device to read in its own time. The size of this queue depends on the implementation, in this emulator it may store up to 32 items and writes following this are discarded. If a device needs to read a lot of data, direct memory access is preferable.

Reading from a port involves another queue just like writing, except the data travels the other way. Reading when there is nothing in the queue will just return the last value that was read, over and over. If a device needs to send a lot of data to the CPU, direct memory access is preferable.

Memory controller

The memory address of 0, physical or virtual, is considered a null pointer and dereferencing it will cause an exception. Pages of memory are 4 KiB = 4096 bytes wide, and one address corresponds to a single byte.

When the CPU is running in virtual memory mode, the memory controller implicitly performs a translation upon every memory dereference to convert the virtual address into a physical address - looking it up in the Page Directory / Page Table. It is recommended to read section 3.3 first, which provides the software-side explanation and defines some useful initialisms.

The procedure for the translation of a virtual address into a physical address is as follows:

- Raise null pointer dereference exception if virtual address is `0x00000000`
- Get address from PDBR - this is the PD base
- Add the most significant 10 bits of virtual address, multiplied by 4

– Dereference that address to retrieve PDE

– Mask off the least significant 12 bits - this is PT base

– Unpaged address fault if retrieved address is `0x00000000`

– Add to PT base the next most significant 10 bits of the address, multiplied by 4

– Dereference that address to retrieve PTE

– Mask off the least significant 12 bits - this is page base

– Unpaged address fault if retrieved address is `0x00000000`

– Add remaining 12 bits of the address to page base - this is physical address

The CPU may query the memory controller as to how much RAM is installed in the system. It does this by sending the ID 1 (an unsigned 32-bit integer equal to the number 1) via the IO port to the memory controller, and waiting for the controller to raise interrupt 16 - metadata available. Then, it may read from the IO port an unsigned 32-bit integer indicating the number of physical pages present - memory is always sized as a multiple of 4 KiB.

Serial out

This is simply a one-way channel for 32 bit values to be written out by the CPU - so the device connected is free to interpret these however it wants. In this implementation, the least significant byte of each value is taken as a Latin-1 encoded character and printed by the emulator to standard output immediately.

Disk

The disk device is a store of 512-byte chunks called sectors. The emulator is agnostic to the disk format: even the loading of the first sector (which we've designated as the bootloader) is done by writing ROM code rather than it being baked into the implementation itself. The disk device provides two operations - reading a sector and writing a sector.

To read from a sector, write to the disk's IO port (using `OUT`) twice: first with the sector number (which is 0-indexed), followed by a physical address in memory of where it is to be read into. The disk completes this operation in its own time, and it is advisable for the program to `HLT` until the disk raises interrupt "Data read into memory".

To write to a sector, write to the disk's IO port twice: first, with the sector number, but with the setting of the MSB / ORing with 0x80000000. This is followed by a physical address in memory of where it is to be written from. The disk completes this operation in its own time, and it is advisable for the program to `HLT` until the disk raises interrupt "Data written to disk".

Keyboard

The keyboard device has the job of sending key-codes (which must be translated to ASCII based on the keyboard layout) to the processor. When a key is pressed, the keyboard

device raises the interrupt "Key pressed". This signifies that there is a new piece of data waiting in the keyboard's IO port, to be read into the CPU.

Upon reading from the port using `INP`, the value will correspond to the code of the key that was pressed.

Display screen

Since outputting to the screen requires the communication of large blocks of data at a time, it operates using direct memory access. The display accepts an output instruction with a pointer to a page in memory that contains the contents of the screen to be drawn - 4096 bytes = 32768 pixels (one bit per pixel, 32 bytes per row) in a 128 rows by 256 columns screen. If the bit is 0 the pixel is to be drawn black, otherwise it will be drawn white. The address supplied using `OUT` must be the physical address in memory, as the memory access does not go via the CPU and is not translated.

The top left corner is the most significant bit of the byte at offset 0. The next bit along in the byte at offset 0 is the pixel to the right, and so on. The top right corner is the least significant bit of the byte at offset 31. The most significant bit of the byte at offset 32 is the pixel directly below the top left corner. Left to right then down, most significant bits to least significant bits.

After telling the display to draw from a page in memory, that page should not be modified until a new pointer is sent to a different area. The CPU will not know how long it takes for the display device to copy out those bytes to its internal buffer, nor know when it has finished - so to prevent corruption only update the pointer when the whole frame is ready and then don't touch that memory while it's on-screen.

2.3 Initial ROM execution

When the processor is started, all registers (including special registers) are set to 0. Memory is also 0 at start-up. The ROM code would, in theory, be in the ROM chip on the motherboard - here it is just provided by the emulator. The first action of the processor is to copy the ROM code (about 60 bytes) into memory, starting at the address 0x10. Then the processor sets `IP`=0x10, thus jumping to the beginning of the code, and commences execution. Virtual memory and interrupts are disabled.

In this OS, the ROM has three tiny stages. First, it sets the stack pointer to a safe value of 0x1000, which creates the (temporary) stack growing backwards into the first page of memory. It also writes the IVTR special register to place the interrupt table in the second page of memory - beginning 0x1000. Second, it registers an empty interrupt handler for "Data read into memory", and requests that the disk writes the first sector into memory starting at address 0x100. Finally, it `HLT`s to wait for the operation to complete. After the empty interrupt handler runs, it jumps to address 0x100 and booting continues with the bootloader.

2.4 The assembler

Assembly syntax

In an assembly language format, each line holds exactly one of the following properties: it is blank (or comment), it is an assembler directive, it is a label, or it is an instruction. A semicolon `;` denotes the start of a comment: everything up to the end of the line, and including the original semicolon, is ignored. This means that comments can be used either inline or on their own line, and in the latter case the line is considered blank.

All instructions begin with specifying the required opcode by using its mnemonic (see section 1.3). Prefixes that inform the processor to use the 8-bit or 16-bit variants of the instruction are concatenated directly after the mnemonic, without whitespace - for example `zrx.8`, which zero-extends an 8-bit value into a 32-bit value. If the instruction takes no operands (e.g. `hlt`, flag-setting operations, etc) then the instruction is complete at this point. Otherwise, at least a single space character must be used after the mnemonic, and then one or two operands can follow.

If two operands are used, the source written before the destination and a comma separates them. Whitespace between the comma and an operand is optional. Operands are written in the same way as the "Meaning" column in the table in section 1.1, where `r` is replaced by any valid register in the table at section 1.2. Integers, such as `immX` and `uimm32`, can be written in decimal, hexadecimal prefixed with `0x`, octal prefixed with `0o`, or binary prefixed with `0b`. Other than making these two substitutions, all other characters and symbols in the "Meaning" column must be used verbatim.

An integer can also be written by using a label (in which case it refers to a memory address). Label names must start with a fullstop `.` and may not include whitespace. A label must be defined by placing its name, followed by a colon `:` , on the line immediately preceding an instruction - and when the file is assembled, any references to the label will be replaced by the address of that instruction. Thus, labels can only safely be used to substitute a `uimm32`.

The following example of correct assembly code for this processor demonstrates these features:

```
    cpy [bp + 8], im
    and 0xfffff000, im      ;   page base address
    cpy zr, kx
    dec kx
.zeropage_loop:
    inc kx
    cpy zr, [im + kx*4]
    dsub 1023,kx
    jnzr [.zeropage_loop]
```

`bp`, `im` and `kx` are registers that are being used in memory references. `.zeropage_loop` is a label that's being used to create a while-loop construct: its address is being used in the instruction `jnzr` which is a conditional branch. Integers are being specified using both decimal and hexademical, and also note that lines containing an instruction have been indented - adding whitespace to the beginning of lines is legal in this assembly language and can improve readability. We also have an inline comment that just gets ignored by the assembler.

Assembler directives are commands for the assembler, not the processor. Labels are a type of assembler directive that have already been covered, and in a similar vein one can use

```
$varname "contents"
```

to insert a null-terminated ASCII string containing `contents` at the very end of the assembled file, and cause any reference to the string-label `$varname` to translate to the memory address of its first character. String-labels must begin with a dollar sign `$` .

The assembler has to know the address that the program will be loaded to in memory (the address of the first byte of the binary once it's in the processor's memory), because it has to ensure it calculates the right offsets for labels. We direct it so using one of:

```
# 0x1337
#+ 0xE000
```

The first one is using an absolute address. There must be one of these at the start of every program, and it again tells the assembler the address that the program is going to start at - in this case the address of 0x1337. The second one is relative though, it can be used halfway through a program to instruct the assembler to add that number to every label it sees going forwards. This is particularly useful if code has to relocate itself halfway through. In this case, the assembler would add an offset of 0xE000 to all labels and strings, all code, after the directive appears. Again, the symbol for an absolute loading address is a hash `#` and for a relative change it's also followed by a plus, `#+`

The assembler supports splitting code into multiple files, which can be used to make pseudo-libraries. The way that these included files of code are marked is that their filename must begin with an underscore `_` . And as such, the symbol for making an inclusion is also an underscore, so one just writes the name of the file into their assembly code:

```
_fn_zeropage.txt
```

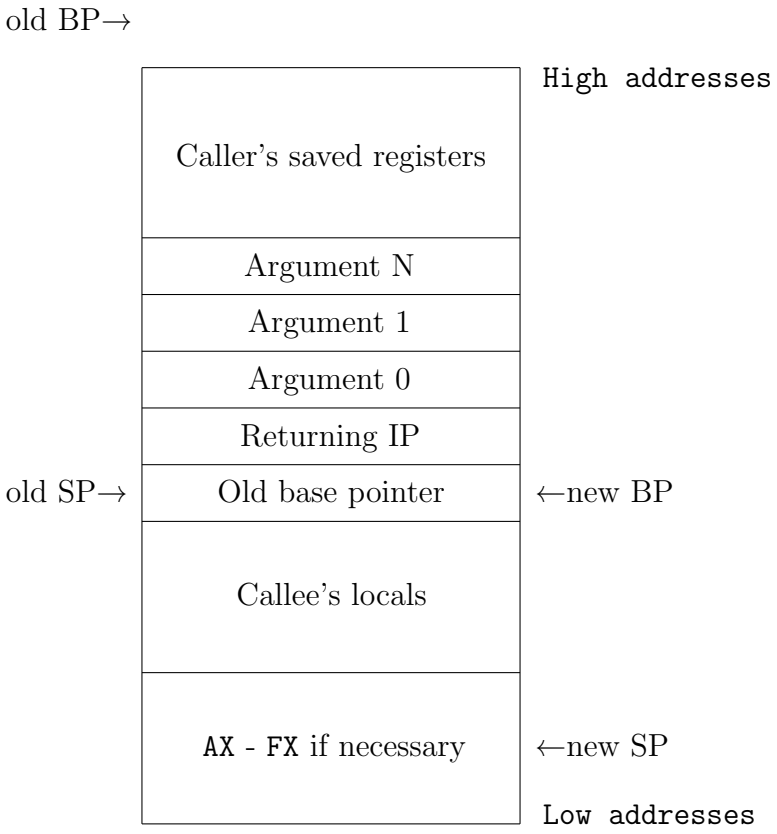
And then the assembler will take the contents of the file (in the same folder) called `_fn_zeropage.txt`, and insert it into the assembly file it was included in. For this reason, it is essential to avoid using the same label name in two files, where one includes the other. After including these files, all the assembler sees is one long assembly program.

Calling conventions

To call a function:

- The caller pushes any registers that it wants to save, that aren't saved by convention, onto the stack. Some general purpose registers are preserved by convention, plus `SP`, `BP` and `IP`.
- The caller pushes any arguments to the function onto the stack
- The caller executes a `CALL` instruction that pushes the returning instruction pointer and jumps to the address given
- The callee pushes the old base pointer, and then sets the (new) base pointer to the stack pointer. The first item of this new stack frame is therefore the saved old base pointer

- The callee subtracts some amount from the stack pointer to allocate space for its local variables
- It is the callee's responsibility to preserve the six general purpose registers when it returns that are preserved by convention: `AX` through to `FX` inclusive. If it doesn't touch them then that's fine, but if it does, it should push them to the stack using `PUSHR` and `POPR` to restore
- If the callee wishes to return a value, it should set the implicit register `IM` with either the value or a pointer to it
- At the end of the callee's execution, it sets the stack pointer to the base pointer and pops the stack into the base pointer. This restores the old stack frame
- It then executes a `RET` instruction which pops the stack (the value at the top of which is the returning instruction pointer) into the instruction pointer, and thus control is handed back to the caller



This diagram shows the generic stack layout when calling a function.

Operation of scripts including `assemble.py`

The following scripts are written in Python, and assist the user in preparing the state for the CPU. They should be run with a version of Python ≥ 3.10, and all without arguments: for example, running the command `python assemble.py` to run the assembler.

`assemble.py` is used to convert all the assembly files in the directory `source_files/` into a binary format that the processor can execute, and place (most of) the results in the

directory `disk_files/`. The one exception is that if a file called `ROM.txt` is present in the source directory, the assembled result will be placed outside the `disk_files/` directory (on the same level as it) and called `ROM.bin`. This is because ROM code, which is the first thing the processor executes, is not stored on disk - instead the emulator reads it in directly from its own file. Otherwise, every assembled file has the same name as the source file, with `.txt` replaced with `.bin`.

Files with names that begin with an underscore `_` are not directly processed as standalone files. `assemble.py` skips them when looking for assembly files that should be assembled into the output directory, but may include them only as part of another assembly file that is directly processed. This recursion only happens once - an included file may not itself include files.

`assemble.py` processes a file in two passes. The first pass reads an input file line-by-line, going out to disk whenever a file inclusion occurs and simply inserting those lines in place of the inclusion marker. All operations to determine what's in a line are preceded by stripping extraneous whitespace from the beginning and end of the section of the line, so generally any amount of whitespace before, between or after the components of an instruction is permissible. Assembler directives are processed at this point too, and the assembler generates a map of label names and their corresponding addresses. This happens because, in the first pass, the assembler keeps track of the width of every instruction it processes, and can then accurately calculate the offset of any instruction from the first. Combining this with the loading address directive, it can translate any label into an address on first pass.

Instructions are also converted from their text form into an instruction object, represented by an array with 6 values - 8/16/32-bit mode, opcode, source type, destination type, values used in the specification of the source, values used in the specification of the destination. For example, a destination specified by `[0x1337 + ax + bx*4]` would consist of values `[0x1337, 1, 2]` (because register `AX` has code 1, `BX` has code 2) and the `*4` is actually specified by the destination's *type* - in this case 0b1101 (see section 1.1).

In the second pass, the actual resultant binary format is built. Since instructions are now in a convenient format, `assemble.py` iterates through the stream of them and now has the ability to replace labels with their addresses after creating the mapping in the first pass. The instructions are correctly padded and written to the output file, followed by any strings that were declared in the program.

`generate_font.py` is a small script that reads a bitmap font file, `font.bmp`, and writes each pixel left to right then top to bottom into a file. That is to say, it flattens the 2D-array of pixels in the image. The resultant file is written as `disk_files/font.bin`.

`generate_keymap.py` is a helper script to generate a lookup table of keyboard scan codes to ASCII codes. It reads a file called `keycodes.txt`, in which each line contains a keyboard scan code, followed by the non-shifted version of that character, followed by the shifted version of the character. For example, the line

```
10 1 !
```

indicates that when keyboard scan code 10 is seen, it corresponds to the ASCII numeral 1, unless shift is pressed in which case it corresponds to an exclamation mark. `generate_keymap.py` writes the file `disk_files/keymap.bin` as an array of bytes, in which `keymap[2*k]` retrieves the non-shifted character for scan code `k`, and `keymap[2*k + 1]` retrieves the shifted character.

`generate_disk.py` creates a disk image file, called `diskfile.img`, from the binary files in `disk_files/`. First, a file called `BOOT.bin` *must* be present in the disk files directory, for it then is end-padded with 0s until 512 bytes long and used as the first sector of the disk. (Sectors are just stored sequentially in the file). Then, for all other files up to a maximum of 16, `generate_disk.py` appends their contents starting from the third sector and makes an entry in the filesystem table at the second sector. The extension is removed from the filename, and this becomes the name that the file takes on the CPU's disk.

2.5 The emulator

The emulator is written in C, and loads two files as input - the initial ROM code, and the disk image file. The emulator may be compiled with gcc 12.2.0 on Unix systems with X11 using the command:

```
gcc channel.c disk.c window.c main.c -pthread -lX11 -flto -O3
```

The last two flags enable optimisations and are not strictly necessary.

Startup

To begin, the emulator allocates a large area of memory (1 GB) to serve as the RAM for the processor. However, it uses the syscall `mmap()` to do this, which means that the host system will only use the pages that are actually in-use: and doesn't need to immediately allocate and start using the entire region. Using this method also guarantees that every page of memory is initialised to 0. The display and disk devices are started in their own threads, and then the ROM code is read from `ROM.bin` and copied into the CPU's memory starting at address 0x10. All flags and registers are initialised to 0, save for the instruction pointer which is set to 0x10 and then execution begins.

Main emulation loop - `main.c`

First, any exception that was generated during the previous cycle is handled if interrupts are enabled (if `IEF` is set). This involves saving the required state information and setting the instruction pointer to an address looked up in the processor's memory (see section 2.1 for a full explanation). If an exception occurs with interrupts disabled, the emulator immediately terminates as there is no way to recover. Other cases where interrupts are enabled, but an exception is caused during the handling of the interrupt, the handler is run again for the exception. The way that exceptions are implemented is by using C's `longjmp()`, so that execution can immediately be whisked away from the instruction that caused the exception, and have the return pointer point to it and not the instruction following.

Then, the processor prepares to read the instruction starting at the instruction pointer. This byte is read, and if it indicates a prefix then the next byte is read too - so that the opcode has been determined as well as the presence of the prefix or not. Exceptions may occur due to an unrecognised opcode. Once the emulator knows the opcode it knows how many operands to read, and if the instruction takes no operands, it executes it there-and-then. One interesting instruction with 0 operands is `HLT`, where (if interrupts are

enabled) the main emulator loop blocks until it receives an interrupt on a thread-safe queue, which all emulated devices running in other threads can push to.

The next task of fetching the operands forks in two depending on whether the instruction takes one or two of them - but the two-operand version is largely the code for the one-operand version but with two of everything, so the process is the same. In largely the reverse as how the assembler generates its binary code, the operand type is read and used to determine how many nibbles should be read for the actual components of the operand. Various exceptions may occur if the combination of operands and the opcode is illegal. The backbone for the process of fetching *the operand value* from the operand specification is a lengthy function containing a `switch` statement over all the operand types.

Once the emulator knows the actual value that the operand refers to, or the values for two operands, it can execute the instruction. While most instructions involve reading the destination value, combining it with the source value if present and writing back to the destination, sometimes this is not followed and the emulator can make optimisations by skipping various fetching and writing operations. The amount of duplicated code in the execution of each instruction is kept to a minimum by having any operands loaded as resolved values before the large `switch` statement, and placing the result in a temporary `result` variable rather than having to figure out where to write it to.

Writing the result (if it needs to be written) is done at the end of the loop, along with checking for hardware interrupts via thread-safe queues. If it has been decided that a branch condition is true, the `IP` register is set to the destination operand address (as detailed in section 1.7). Flags are recombined into the overall register as well, and the whole loop repeats.

The display device - `window.c`

This code runs an X11 window to which output that the processor sends to the display device is displayed, and also from which keyboard events are caught and sent *to* the processor as interrupts. When it starts at the beginning of emulation, the window is created and made able to show black and white pixels, as that's all the display device supports at this time (section 2.2). Certain X11 events must be handled, like when the window becomes "exposed" - the window must be redrawn if this happens. The code running the display remembers what's on the screen by storing a buffer of rectangles (large pixels) that can just be fed to an X11 library function that draws them.

This rectangle buffer is updated whenever the main emulation loop signals to the device that an `OUT` instruction has supplied data to its port, and so another thread running as part of the display code goes off to copy the screen region out of processor memory and translate it into a series of square pixels (our rectangles), otherwise known as direct memory access. A re-draw is also triggered after the rectangle buffer has been updated.

When a key is pressed, its scan code is sent to the CPU from the keyboard port, ready to be accepted by an `IN` instruction. An interrupt is also triggered, which is how the program running on the CPU knows to read from the port. These interrupts are sent down one queue, which can be accessed by any device in any thread, and is thread-safe for multiple concurrent pushers.

The disk device - `disk.c`

This code emulates the disk device. When it starts at the beginning of emulation, the file called `diskfile.img` is opened and its contents loaded into memory. From there, the disk's event loop blocks until it has received two values to its port - specifying the sector to read, and the address to copy to/write from. The operation is performed using direct memory access, similarly as in the display device, and then an interrupt is triggered to notify the CPU that it has finished.

Chapter 3

The OS

3.1 Filesystem

The disk is modelled by a binary file passed as an argument to the emulator. To make things simple, blocks and sectors are the same size (512 bytes) and are both linear and identity mapped - the first block is the first sector, and all sectors reside contiguously in the "disk" file. The first sector also happens to be the boot sector (containing the bootloader), which is loaded by the ROM and jumped to. Boot sectors in this OS end with `1a f9 49 33`.

The filesystem that the bootloader expects is implemented as a flat filesystem, with a maximum of 16 files. A filename may be up to 23 characters long. The second sector on disk contains an array of fixed-width structs that specify essential and basic metadata about the files resident - known as the "filesystem table".

Pos.	Name	Width	Format
0	Filename	24 bytes	null-term ASCII
1	Start sector	4 bytes	i32
2	Length in bytes	4 bytes	i32

Each struct is therefore 32 bytes, and 16 structs will be present in the array in the sector. If a file is not present, the filename field must begin with `0x00` (null) - i.e. a blank filename. These structs can be thought of as "slots" for a file to be in - if the disk has fewer than 16 files, some slots must be marked as empty in that way.

3.2 Booting process

Bootloader

The bootloader is the next step in the chain from the ROM. The job of the bootloader is to more cleverly read the filesystem on the disk (by examining the second sector) and determine which sectors need to be read in order to load the kernel (however long it is). It copies these sectors contiguously starting from address 0x2000, making 0x2000 the correct address to jump to, once the bootloader finishes, to run the kernel. Notice that the CPU is still operating in physical memory mode - the bootloader should be agnostic to whether or not the kernel turns on virtual memory (paging), so it is the kernel's job to enable virtual memory if it so desires. The bootloader finds the correct filesystem entry

for the kernel by comparing each name string field to "kernel", followed by a 0 byte to terminate. The bootloader simply quits if the kernel is not found.

First kernel tasks

First, the kernel needs to enable virtual memory paging. This task is somewhat complicated because the processor will begin translating memory addresses as soon as the flag is set, so not only do the page tables need to be properly set up, but also the next address needs to be valid in virtual memory as there isn't time to perform a `JUMP` in-between.

The feat is accomplished by setting up *identity paging* for all the pages in-use in memory that the kernel knows about. So, it first chooses the next four available pages: one for the page directory (the address of which is entered into the PDBR), for the first page table (at index 0 in the page directory), one for the new kernel stack, and one for the first page of the free-physical-page indicator (see section 3.3). Then it sets up the page directory and table to "map" the addresses of the pages containing all the kernel code (plus the new stack and free-physical-page indicator) to themselves, so that turning on virtual memory has no real effect on pointers pointing to anywhere in these pages.

Note - "mapping" an address refers to the process of creating valid entries in the page table, and the page directory if required, to cause the CPU to start translating a virtual address into its corresponding physical one. After this, it ensures that the free-physical-page indicator accurately reflects the addresses of physical pages that are in-use by setting the bits that correspond to those addresses. This stops the kernel from, later on, forgetting about the physical pages that are behind the virtual mappings and overwriting them.

Then, by executing `SETVMF`, the kernel enables memory paging. The CPU now treats all memory addresses as virtual, and uses page directories and tables to look up mappings to physical addresses. At this point, it is advantageous to relocate the kernel up to a very high address of 0xC0000000, and then all user-mode address spaces can reside in the lower addresses. But because all of our addresses are now virtual, all this constitutes is allocating a couple of page tables for 0xC00... and 0xC04... addresses (the first for code, the second for data) and then doubling the code mappings from their identity maps into 0xC00. The kernel also sets `BP` and `SP` to use the first page of kernel stack that we reserved earlier on, though at this point those are still identity-mapped addresses (again, meaning that the virtual address is the same as the physical address).

Higher-half kernel tasks

After jumping up to the high memory address (made possible by telling the assembler that a new loading address has been used), the kernel's duties largely involve setting up - allocating and mapping pages for - the state that it needs to its job with syscalls. These state objects all reside in 0xC04 addresses, according to the following layout:

Address	PD[i]	PT[j]	Description
0x00001000	0	1	user code
0x00400000	1	0	start of heap
...			
0xBFFFF000	767	1023	initial user stack pointer, stack grows backwards
0xC0000000	768	0	kernel code, service routines, functions
0xC0400000	769	0	frame buffer 0
0xC0401000	769	1	frame buffer 1
0xC0402000	769	2	process metadata stack
0xC0403000	769	3	interrupt vector table (physical address = 0x1000)
0xC0404000	769	4	temporarily mapped during `exit()`
0xC0405000	769	5	temporarily mapped during `spawn()`
0xC0406000	769	6	cached file system table
0xC0407000	769	7	kernel disk read buffer
0xC0408000	769	8	font file (can be up to 8 pages long)
...			
0xC0410000	769	16	keymap
0xC0411000	769	17	string input buffer
0xC0412000	769	18	string creation space
...			
0xC041E000	769	30	kernel main stack (pointer initially C041F000)
0xC041F000	769	31	kernel interrupt stack (pointer initially C0420000)
0xC0420000	769	32	free physical page indicator 0
...			
0xC043F000	769	63	free physical page indicator 31
0xC0440000	769	64	variables

Some of these objects, such as the free physical page table number 0, already have a physical address since we needed to create and access them before the kernel turned on virtual memory. So the kernel maps them manually (also including the interrupt vector table, and the kernel main stack which has its current virtual address still equal to its physical address, in `BP`). The rest use a nice "automap" function which scans the free-physical-page indicator for the next available one, and calls out to map it itself.

The kernel can then remove all the old identity (virtual = physical) mappings, so that it lives entirely in the high addresses in memory. It does this by filling the first page table in the page directory (for addresses beginning 0x000...) with 0s, clearing it, and then marking it as available. Then, variables are initiated in the "variables" page, and all the exception/interrupt/syscall handlers are registered by moving their addresses to the correct offset in the interrupt vector table. The kernel also loads in a cache of the filesystem table by accessing the disk, and uses that to read in the font file and keymap from disk. The loading of those two files, which are used to draw received keypresses to the screen, concludes the setup tasks. The kernel is now ready.

The kernel now calls `spawn()` for a file called "init", loading and executing the first user-mode process. It runs this process until it dies, and then exits itself causing the processor to halt.

3.3 Paging

Table structure

Memory pages are 4096 bytes long, and aligned to 4096 bytes too.

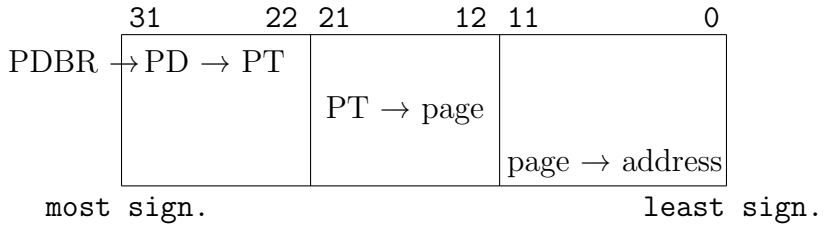
The PDBR (page directory base register) is a special register that stores the physical base address of one page of memory. Located in this page is a Page Directory that the CPU will use to translate virtual memory addresses into physical ones. The contents of this register can be changed during a context switch so that each process can have its own memory mappings from its own Page Directory.

The Page Directory is an array of 1024 4-byte objects, called PDEs (Page Directory entries). Each PDE refers to a Page Table for the set of virtual addresses in the range of the PDE. It is important to note that, since pages are aligned, we only need the most significant 20 bits of a 32-bit address to point to the start of a page. Hence, the first 20 bits of the PDE point to the *physical* address of the Page Table, and the last 12 bits do nothing and are reserved for future flags.

The PDE that gives the address of the right Page Table for a given virtual address is indexed by the most significant 10 bits of that virtual address. Because each PDE is 4 bytes, this index must be multiplied by 4 and added to the PDBR - or, combining both steps, the virtual address right-shifted by 20 (by 22, but then SHL by 2), then the least significant 2 bits masked off, and added to the PDBR. After dereferencing into the Page Directory, we now have the physical address of a Page Table - and the next step is very similar to this one.

A Page Table is an array of 1024 4-byte objects, called PTEs (Page Table entries). Each PTE refers to a page of memory, that is the result of the translation. Again, we only need the most significant 20 bits to point to the page, so the last 12 bits do nothing and are reserved for future flags. The PTE that gives the address of the right page is indexed by the *next* most significant 10 bits of the virtual address - that is to say, bits 21-12 inclusive if least significant bit is indexed 0. A similar procedure as above can be followed to obtain the address of the page - shift the virtual address right by 10, mask off the least significant 12 bits and add to the base address of the Page Table.

To see what the hardware does with these tables that we've just set up, please refer to see section 2.2.



This diagram shows how the first 10 bits index into the Page Directory, the next 10 bits index into the Page Table and the last 12 offset into the page.

Recursive page mapping

On allocating a new Page Directory or Page Table, the *entire* page needs to be zeroed so that, as all indexes are not records yet, they all point to the null pointer and would cause an unpaged address/null pointer dereference exception. But, importantly, the last

entry in a Page Directory/Table must map to the physical address of the base of the page it's in. This allows the kernel to actually modify the tables, as it can perform a sort of 'recursion' to index not into the final page but either the PT or the PD.

To modify the PDE at address `0x4D4` in the PD:

`*(0xFFC00000 + 0x003FF000 + 0x000004D4) = 0xDEADC0DE` – hit the last value in the PD, which points to the PD, so now what should be the PT is still the PD. Hit the last value in the PD again, so what should be the actual page is still the PD. Now point to address `0x4D4` (which would be index `0x135`) and modify, and we'll hit the physical address of that entry.

To modify the PTE at address `0x4D4` in the PT for addresses beginning `0xE4800000` (`1110 0100 10...`):

`*(0xE4800000 + 0x003FF000 + 0x000004D4) = 0xDEADC0DE` – hit the PDE for that range of addresses, which points to a PT, as normal. Hit the last value in the PT now, so what should be the actual page is still the PT. Now point to address `0x4D4` and modify.

The same trick can be used to get the physical address (as a value) of any PD or PT, just recurse once more at the end.

Free pages

The kernel also maintains a global `[# of physical pages]`-bit area of memory with each bit indicating whether the corresponding physical page is free or used. For example, with 2 GiB of RAM, this would be 65536 bytes (pages are 4KiB) which is 64KiB. To find a free physical page to map a virtual page to, the kernel may just read sequential bytes until it finds one with at least one bit not set. The kernel creates this bitmap as one of its startup duties, as it knows about its own pages (the only pages that would be in use by the system at that time), and can find their physical addresses using the methods described above.

3.4 Syscalls

Each of the following "syscalls" denote a special software interrupt number that, when generated by the instruction `GENINT`, cause the kernel to perform a useful task for the user-mode process. If the system call takes arguments, they are listed here in the order that they should be pushed onto the stack in. If the system call returns a value, it will be in the `IM` register. The system call may "clobber" registers, which means that the value in them cannot be guaranteed to be preserved and must be saved by the calling user-mode code.

Spawn - 0x40

`spawn()` - start a process running code from a file. Information about the currently-running process is saved, and then a new Page Directory is created, with kernel mappings duplicated into it - so that all process paging tables share the same kernel maps. The address space is switched, a stack growing backwards from 0xBFFFF000 allocated, and then the filename is searched for in the file system table. If it is not found, the code returned is 0xFFFFFFFF. Otherwise, the code is loaded from the disk and paged into the new address space starting at 0x1000. The system call then jumps to this address and user-mode execution begins, but notice that it is the return of syscall `exit()` that

provides the exit code, in a simulated "return" from `spawn()` (spawn moved to a kernel stack during its execution, preserving the user-mode stack of the code that called it, ready to switch back to during `exit()` and `IRET`).

First argument: 4-byte pointer to a null-terminated filename string to search for

Clobbers: *all registers* except `SP`, `BP`, `IP`

Returns: exit code of process once finished

Exit - 0x41

`exit()` - terminate the current process with exit code. The system call switches to a kernel stack and then iterates through all entries in the Page Directory, and for each Page Table it finds it marks the physical addresses of any mappings in it as free, using the free-physical-page indicator, and zeroes the Page Table. It also marks the Page Table's physical address as free, and eventually can zero the Page Directory and mark its physical address as free. This is to say, it completely reverses all page allocations that the process performed. Then, it switches back to the address space, stack, and state of the previously-running process (that is to say, the process that spawned the one that is currently exiting) and returns - simulating the eventual return of the initial `spawn()` syscall.

First argument: 4-byte return code

Clobbers: doesn't matter, no execution of calling code after calling this syscall

Returns: doesn't return to calling code

Alloc - 0x42

`alloc()` - allocate N bytes on the heap. This system call keeps track of the next available address on the heap, works out the virtual page addresses that the allocation will go through and ensures that they are all allocated (reserving and mapping them if not). Then, it writes the length of the allocation at the very beginning of the run of memory (of length N + 4), and returns a pointer to just after that. The memory is known to be zeroed because whatever last held it has the duty of clearing it.

First argument: 4-byte integer representing the number of bytes to allocate

Clobbers: `IM`

Returns: address of first byte of the allocation

Free - 0x43

`free()` - undo an allocation by `alloc()`. The length of the allocation is found by reading the metadata just before it, and then the run of memory of that length is zeroed. In a future version of this Operating System, this will properly "free up" the memory, but for now it does not actually mark the bytes as being available for other `alloc()`s.

First argument: 4-byte address of the allocation, initially given by an `alloc()` call

Clobbers: nothing

Returns: nothing

Map - 0x44

`map()` - allocate a page of virtual memory. This calls the "automap" function seen previously, which performs two things: first, it scans the free-physical-page indicator to

find the very first page of physical memory that's marked free - and then sets a 1 in its bit position to mark it as reserved. After the physical address of this page has been found, it will create a page table if it needs to (if no other addresses in the domain of this page table have been mapped yet). Finally, it writes the corresponding entry in the page table to allow the CPU to translate the given virtual address to the physical address of the free page that the syscall just found.

First argument: 4-byte virtual address of the page that should become allocated and mapped

Clobbers: nothing

Returns: nothing

Unmap - 0x45

`unmap()` - remove a virtual memory page mapping, i.e. undoing a call to `map()`. First, the whole page is filled with zero so that when it is next allocated it's ready for use immediately. Then, the corresponding entry in the page table is zeroed out - this is the definition of a virtual address not being mapped to anything. Finally, the physical page that was mapped is marked as free, so that's setting a 0 in its bit position in the free-physical-page indicator.

First argument: 4-byte virtual address of the page that should be cleared and unmapped

Clobbers: nothing

Returns: nothing

Print - 0x60

`print()` - writes a string to the display. For each character in the null-terminated ASCII string supplied, a subroutine is called that draws that character to the screen and then refreshes it. Since the kernel remembers where on the screen the cursor is (or the location of the next character), it can easily look up printable ASCII characters in its font file (loaded into memory), and then set the pixels accordingly. For a carriage return/newline, it just invokes the routine for a newline (which may involve "bubbling" the text up the screen to automatically scroll). The same routine may be invoked when the text reaches the end of the screen width, so that the line can wrap. The kernel communicates with the display device by storing two identical copies of the screen buffer, then flipping the pointer that the display device is drawing from each refresh. It can then safely write into the *other* one, as its contents aren't being accessed by the display at that time.

First argument: 4-byte address of the first character of a null-terminated ASCII string

Clobbers: nothing

Returns: nothing

Get - 0x61

`get()` - get user input, blocking until enter pressed. In addition to the kernel handling keyboard events, translating the scan code into an ASCII character and drawing it to the screen (including performing the proper procedure for a backspace), it also stores the resultant string in a buffer. The buffer is cleared when enter is pressed (so that it keeps track of the line currently in progress), but not before the kernel checks to see if

there's a call to `get()` waiting on the result, and if so is able to return the string. The way that this works is that the system call registers a hook with the line buffer section of the kernel, and then halts - once the buffer is ready to go, the interrupt handler simply jumps to the next line of the system call after the halt. There, a complete "return" from the handler is simulated, and the system call can then copy the string into the user-mode code's buffer to return it and return normally.

First argument: 4-byte address of buffer in which to place the resultant string

Second argument: Maximum length of string that may be safely copied to the buffer (truncation after this point)

Clobbers: nothing

Returns: nothing

Flen - 0x62

`flen()` - retrieve file length in bytes from filename. This syscall is very useful as a precursor to `fread()`, allowing user-mode code to allocate a buffer of the correct side to read the file into. It takes the filename and iterates through the filesystem table, string-comparing it to the filename field of each file metadata struct in the table. If the filename is not found, the syscall `flen()` returns 0, otherwise it then returns the number of bytes in the file as reported by the filesystem table for the matched filename.

First argument: 4-byte address of the first character of a null-terminated ASCII filename

Clobbers: nothing

Returns: 0 if file not found, otherwise number of bytes in file

Fread - 0x63

`fread()` - read file contents into memory. After finding the entry (if existent) in the filesystem table in the same way as described in `flen()`, it then calculates how many whole blocks it needs to read and performs those disk reading operations, calling out to a function that actually handles the IO. This function puts the read data into an intermediate kernel buffer, located at 0xC0407000, which is then copied by the syscall into contiguous sections of the user-supplied buffer. This also prevents a buffer over-run at the end of the file, where a whole block will be copied to kernel memory but only the exact number of leftover bytes copied to the buffer. The total number of bytes read is the minimum between the first argument to the syscall and the actual file length - and this figure is returned by the syscall. So, the buffer to read into should be made as big as the first argument to the syscall.

First argument: maximum number of bytes to read from file

Second argument: 4-byte address of the first character of a null-terminated ASCII filename

Third argument: 4-byte address of buffer to read file into

Clobbers: nothing

Returns: 0 if file not found, otherwise number of bytes read into memory