HT2026 Digital Systems notes

Remaining TODOs: 38

Contents

1. Computers ... 2
2. Machine code ... 2
3. Finite width arithmetic ... 5
4. Memory instructions ... 6
5. I/O ... 7
6. Serial I/O ... 10
7. Interrupts ... 11
8. Operating systems ... 13
   8.1. Device drivers ... 15
   8.2. Context switching ... 15
9. Combinational logic ... 17
10. Sequential circuits ... 21
11. Architectural elements ... 24
   11.1. ALU ... 27
12. Building a datapath ... 27
Index ... 28

1. Computers

Definition 1.1: A computer is a device that produces (a representation of) the outcome of a computation, given a (representation of) a problem specification.

Theorem 1.2 (Church-Turing Thesis): There exists a well-defined method for computing something iff there exists a Turing Machine that produces the result.

Remark: This is a bit circular, but it defined both a computer and a computation.

Definition 1.3: A Turing machine is a machine that has an infinite tape and a "CPU" with the following properties:
- "CPU" has a "state" at each time
- "CPU" sits at a location on the tape
- "CPU" can write a symbol in that location
- "CPU" moves 1 step left or right at each time
- lookup table determines movement and next tape:

$$\begin{aligned} \text{tape}(L_t) &= \text{lookup_table}_1(S_t, \text{tape}(L_t)) \\ S_{t+1} &= \text{lookup_table}_2(S_t, \text{tape}(L_t)) \\ L_{t+1} &= \text{lookup_table}_3(S_t, \text{tape}(L_t)) \end{aligned}$$

2. Machine code

Definition 2.1: RISC = Reduced Instruction Set Complexity. Instructions are load/store (register memory) or arithmetic/logic (register register)

Definition 2.2: The Von Neumann Architecture has a CPU and separate memory (with a single address space) that contains both instructions and data.

Operation:
1. Controller fetches an instruction from memory, at the location in the program counter register (`pc`)
2. Decode instruction
3. Execute instruction
4. Increment `pc` by 1, and repeat

In practice:
- State is represented by a voltage in a wire
- Voltage is high or low, mapping to 1 or 0
- A collection of wires encodes a binary number - the number of wires is the width

ARM has 16 registers:
- r0-r7 general purpose
- r0-r3 are handled differently in subroutines
- r8-r12 general purpose but not all instructions can use them
- r13 stack pointer - stores address where surrent subroutine returns
- r14 link register
- r15 program counter
- processor status register - each bit has a specific meaning

Instructions are 16 bits wide. Note that this means the `pc` is actually incremented by 2 each step.

The N,Z,C,V status bits are in `psr`. For the `adds` instructions:
- N set if result is negative
- Z set if result is 0
- C TODO
- V TODO

Assembly language is a slightly more human-readable source for machine code.

Assembly quirks:
- Immediate form: `adds r0, r1, #3` adds 3 to `r1` and stores in `r0`
- `adds r0, r0, #10` also works
- but `adds r0, r1, #10` does not because it has a different encoding when using different registers, leaves fewer bits for the immediate

Arguments to assembly instructions are in reverse order to in machine code.

E.g. `adds r0 r1 r2 0b 0001 100 010 001 000` (adds `r1` and `r2`, result in `r0`).

TODO subroutines

TODO how to read rainbow table

Definition 2.3: RAM is random access memory; it loses state when it loses power. ROM is read-only memory; it can be read to (it has to to reprogram a chip!) but is more difficult to; it can be read as with RAM. ROM does not lose its state when it loses power.

All memory is mapped into a single address space - different types of memory are given different addresses. Even peripherals are given memory locations. For the micro:bit, all memory locations have a 32-bit address:
- 256kb ROM at 0x_0000_0000 to 0x_0004_0000
- 32kb RAM at 0x_2000_000 to 0x_2000_4000
- I/O devices from 0x_4000_0000

When a `hex` file is sent to the micro:bit, this is text-encoded hexadecimal representation of the object file, in little endian (i.e. each set of 2 bytes is reversed) (i.e. rather than `18404770` we get `40187047`). This is because ARM is little-endian.

Definition 2.4: Little endian systems store the least significant byte of a number in the lowest memory address.

Remark: Reasons for having a separate compiler and linker:
- Some parts can be precompiled and only changes files recompiled, then everything relinked
- The compiler only needs to know about the CPU core (i.e. the instructions set) which the linker can deal with the specific chip details e.g. the memory map.

Remark: When calling a subroutine in assembly/machine code, the convention is that parameters are in r0-r3, the return value is in r0, and r0-r3 can be overwritten freely. TODO double check this

TODO assembly subroutine syntax.

In assembly, comments begin with `@`.

Remark: The fetch-decode-execute cycle executes in parallel, such that most instructions only take one cycle to execute. However, branches disrupt the flow so take 3 cycles to execute (when the branch is taken) - this is because the next two instructions are already decoded and now need to be ignored. Load/store instructions usually take 2 cycles.

Remark: The `pc` is read at the decode stage. Therefore, the `pc` is two instructions ahead when the instruction is actually executed.

Remark: Branch target offsets don't need to encode the last bit because instruction offsets are always even. Furthermore, the offset needs to be shifted back 2 instructions, because of the aforementioned parallel pipelining.

Local variables in a subroutine can be stored in the stack, in memory relative to the stack pointer (register `sp`). Memory below in the stack is used by calling subroutines, and should remain untouched; memory above in the stack is free to be used by the stack routine.

By convention, the stack starts at the top of the address space, and grows downwards, so "above" in the stack has a lower address. So memory below `sp` is usable in subroutines.

ARM provides stack-related instructions for:
- incoming arguments >4
- return addresses
- saved registers
- local storage

The stack instructions are `push {REG_LIST}` and `pop {REG_LIST}`. `push` places the registers in order onto the stack and decreases `sp`; `pop` removes values from the stack and puts them in the registers in the listed order, and increases `sp`.

`push` and `pop` can store/restore lower registers. `push` can store `lr` as well, and `pop` can restore to `pc`.

Note that `pop`ping back to the `pc` incurs another 2 cycle penalty.

`push` and `pop` take $n + 1$ cycles to push/pop n registers.

Definition 2.5: The subroutine contract specifies that:

The calling function
- Places first arguments 1 to 4 in `r0 - r3`
- Places arguments ≥4 on the stack (the called function should know how many to expect)

The called function:
- Leaves values of `r4` and up unchanged *on return*
- Leaves the stack pointer `sp` unchanged *on return*
- Leaves the stack below current location (i.e. memory addresses above `sp`) unchanged
- Leaves return value in `r0`

3. Finite width arithmetic

Definition 3.1: n-bite numbers have two interpretations: unsigned and signed. Unsigned integers are in the range of 0-$2^n - 1$, represented as expected. Signed integers are represented using two's complement.

Define twoc : $\{0, 1\}^n \longrightarrow \mathbb{Z}$:

$$\text{twoc}(a) = \sum_{i=0}^{n-2} 2^i a_i - a_{n-1} 2^{n-1}.$$

i.e. subtract the most significant bit (MSB) rather than adding; if the MSB is 1, the number is negative.

Remark:

Two's complement has a range of -2^{n-1} to $2^{n-2} - 1$ (inclusive).

When the MSB is 0, the signed and unsigned interpretations are equal; otherwise they differ by 2^n.

Therefore, the same addition method used on unsigned integers can be used on signed two's complement integers; we only need to interpret the result in the appropriate way.

The `Carry` bit in the `psr` is set if addition, when interpreted as unsigned addition, overflowed, i.e. a carry bit was produced that couldn't be stored in the register.

The `V` bit in the `psr` stands for oVerflow, and is set if addition, when interpreted as signed, overflowed; i.e. if the MSB of the two inputs was the same, but the MSB of the result is different. Overflow is impossible if the signs are different, because the magnitude will always decrease (which will give a valid result because the two inputs must be valid).

The `N` bit is set to the MSB.

Remark: To implement subtraction, we just need to figure out how to negate a number, and we can then use addition.

Let $\bar{a}$ be the binary complement of a, i.e. $\bar{a}_i = 1 - a_i$. Then

$$\text{twoc}(\bar{a}) = \sum_{i=0}^{n-2} (1 - a_i) 2^i - a_{n-1} 2^{n-1}.$$

Note that negation is always correct *modulo* 2^n.

Then $a - b$ can be computed by adding a and $\bar{b}$ with an initial carry of 1.

Remark: The `cmp` instructions just does a subtraction without putting the result in a register; it just sets the status bits.

We can then try to figure out what all the conditional branching code mean:
- `eq` branches if equal; if `Z`
- `ne` branches if not equal; if `!Z`
- `lt` branches if less than - so either the result is negative and there was no signed overflow, or the result is positive and there was a signed overflow; so branches if `N != V`

4. Memory instructions

There are two main instructions for interfacing with memory: `str` and `ldr`.

Definition 4.1: These have different ways of calculating the address; these are called addressing modes.

- `xxr rd, [rn, #imm]` - loads/stores the value of `rd` from/at the location `rn + imm`; lower registers only
- `xxr rd, [rn, rm]` - loads/stores the value of `rd` from/at the location `rn + rm`; lower registers only
- `xxr r0, [sp, #imm]` and `xxr r0, [pc, #imm]` - special encodings so we can have larger immediates

Addresses must be aligned to the word size (4 bytes), otherwise the instruction will trap.

Native ARM also has a `xxr rd, [rn, rm, LSL #imm]` variation, but this doesn't exist in the thumb encoding.

To get local variables using the stack, we decrease the stack pointer, leaving some bytes free for us to play with, and at the end of the subroutine we restore the stack pointer.

Global variables are stored at a fixed absolute location. But there is no instruction for loading/storing at an absolute location - so we need to put the address in a register. However, the `movs` instruction doesn't allow directly moving a 32 bit constant into a register - this would take several cycles to do. Instead, we put the address into the bytecode, and access the location of the address relative to the `pc`; then we can directly access the memory location. We can't necessarily just access the real data relative to the `pc` as the offset will be too large if the data is in RAM (which it needs to be, if we want to mutate it!).

Remark: `ldr rd, [pc, #imm]` rounds `pc` down to the nearest multiple of 4 before adding the offset.

Example:

```
int count = 0;
void increment(int n) {
  count += n;
}
```

```
ldr r1, =count    @ handy shortand for ldr r1, [pc, #imm]
                  @   (gets the address from ROM)
ldr r2, [r1, #0]  @ load value of data of count from RAM
adds r0, r2, r0
str r0, [r1, #0]

@ then telling the assembly what =count refers to:
    .bss          @ Place the following in RAM
    .balign 4     @ Align to multiple of 4 bytes
count:
    .word 0       @ Allocate a 4-byte word of memoty
```

Arrays are very similar; we just need to allocate more space. For global variables, that means we use `.space <bytes>` rather than `.word <val>`.

Other load/store instructions:

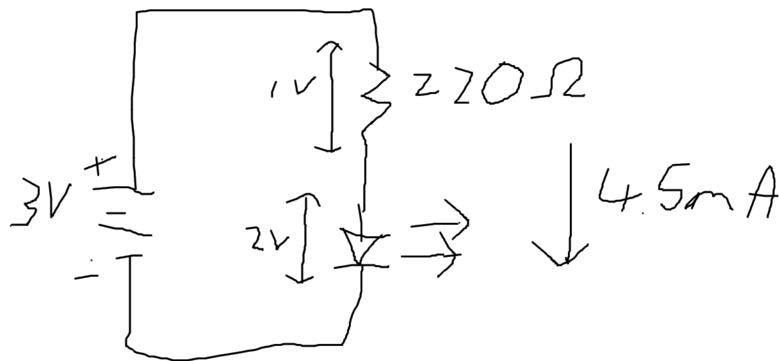
- `ldrb`, `strb` for 8-bit values (uses low 8 bits in register) (useful for strings)
- `ldrh`, `strh` for half-word (16 bits)
- `ldrsb`, `ldrsh` loads 8-/16-bit values with sign extension. This is needed because normal arithmetic instructions only operate on 32-bit signed integers. These fill all of the high bits with the MSB of the loaded value so that the 32-bit interpreted value is correct.

5. I/O

Definition 5.1: The General Purpose Input Output (GPIO) system maps memory addresses to a physical action.

LED design:

Resisters follow Ohm's law: $V = IR$. Suppose that the LED starts emitting light at about 2V. We want the current to be 4.5mA. Then $3 = IR + V_{\text{LED}}$.



$$I = \frac{3 - V_{\text{LED}}}{R}$$

This is a line with a negative gradient; we just need to choose the correct resistance such that the line crosses the LED's IV characteristic at $2\text{V}, 4.5\text{mA}$.

The GPIO pins act as the voltage source.

Definition 5.2: LED multiplexing controls a large number of LEDs with fewer pins that the number of pins. LEDs are grouped into "rows" and "columns", sharing the same input/anode (+ve) pin or output/cathode (-ve) pin, respectively.

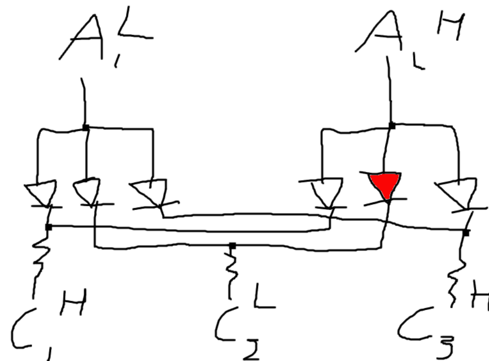


Figure 3: A small example of LED multiplexing, with 2 rows/inputs/anodes and 3 columns/outputs/cathodes

Then if the anode pin is high and the cathode is low, the corresponding LED will be lit; any other configuration results in either zero or negative voltage, meaning that the LED will not light.

LEDs aren't independently controllable, but we can still make arbitrary patterns by switching them on/off very quickly. However, within one group we can still control every LED independently.

On the micro:bit, there are 25 multiplexed LEDs, in 5 physical rows and columns, but electronically arranged in 3 groups of 9 (with the second group having only 7). Refer to schematic for details, but which LEDs are in which group is not sensible (but is kind of symmetric which is nice).

Remark: Peripherals are memory mapped, so to communicate with GPIO, we write a particular value to a specific location in memory.

The two relevant "registers" (memory locations) are `GPIO_DIR` and `GPIO_OUT`. `GPIO_DIR` controls the direction of the pins; `GPIO_OUT` controls the high-/low-ness of the pins.

Example: In assembly:

```
ldr r0, =0x50000504 @ load the address of GPIO_OUT
ldr r1, =0x5fb0     @ load the bit pattern for the LED pins
str r1, [r0]        @ configure the GPIO pins
```

In C:

```
#include "hardware.h"
...
GPIO_OUT = 0x5fb0;
```

`GPIO_OUT` is defined with a macro that expands to `(* (unsigned volatile *) 0x50000514)`, i.e. a pointer to a volatile (can be changed from outside the code) unsigned int at the relevant memory location.

So equivalent to

```
volate unsigned int* a = 0x50000514;
*a = 0x5fb0;
```

The `GPIO_OUT` bit pattern is `...|R3 R2 R3 C9|C8 C7 C6 C5|C4 C3 C2 C1|0000`.

So to light LED 3 in row 2, write `0b0101_1111_1011_0000 = 0x5fb0`.

To display a pattern using all the LEDs, alternate between 3 patterns with a small delay (e.g. 500 µs) between (for a total time of e.g. ≈ 500µs, giving 67 fps). To delay, execute the relevant number of `nop`s to wait for the correct time, based on the clock speed of the CPU. Note that this isn't a great idea because it's not portable, and might break if the chip or compiler changes!

Remark: The circuit for pushbuttons uses a pull-up resistor (more on this later).

To read from buttons in C: TODO

6. Serial I/O

Definition 6.1: Serial communication is one bit at a time (serial) over 1 pin/wire, bidirectional.

The serial contract is:
- keep voltage high when idle
- give a start bit (low) to indicate the start of transmission
- data is read (and sent) at regular intervals (104 µs) - low for zero bit, high for 1 bit
- stop bit: return to high for idle
- then repeat from start bit for another byte
- sender and receiver agree on baud rate - how many bits per second (typically 9600 bits per second)
- must synchronise together with clocks that don't drift

This can be implemented with GPIO, assuming the clock speed is very accurate.

Definition 6.2: On the Nordic chip, we get special hardware to handle serial communication for us, called the Universal Asynchronous Receive Transmit (UART) interface. This provide:
- Full-duplex operation (separate receive/transmit wires, for simultaneous receive/transmit)
- Universal (can operate at different speeds)
- Asynchronous - no shared clock signal (c.f. SPI)

Example: A basic UART driver setup:

```c
void serial_init(void) {
  UART_ENABLE = 0;
  UART_BAUDRATE = UART_BAUD_9600; // set the baud rate with magic constant from
datasheet
  UART_CONFIG = UART_CONFIG_8N1; // 8 data, 1 stop (i.e. 1 byte at a time)
  UART_PSELTXD = USB_TX; // choose pins
  UART_PSELRXD = USB_RX;
  UART_ENABLE = UART_Enabled;
  UART_RXDRDY = 0; UART_TXDRDY = 0;
  UART_STARTTX = 1; UART_STARTRX = 1;
  txinit = 1;
}
```

Transmitting a character:

```c
void serial_putc(char ch) {
  while (!UART_TXDRDY) {/* idle */} // wait for "transmit ready" to become 1
  UART_TXDRDY = 0;
  UART_TXD = ch; // TXD = "transmit"
}
```

This polls to wait for the previous characte to finish transmitting (because hardware deals with sending the individual bits!). `printf` is a wrapper around this.

Remark: UART gives us the opportunity to do something useful while transmission is in progress, but we chose not to here, and it would be a bit of a pain to do anything useful in this setup.

One possible solution is to store characters in a circular buffer, and modify the program to poll `UART_TXDRDY` and send the next character in the buffer when it is ready. But this doesn't give us a modular design, and depending on where we poll, might not respond as immediately as we might like.

If we want to ensure that we don't lose data when the buffer fills up, we can then fall back to waiting in a loop (that polls for UART! otherwise it will get stuck in this loop) until there is room for more characters.

To make the circularness efficient, choose buffer length to be a power of two so that modulo can be implemented as a single bitwise-and.

7. Interrupts

As soon as a hardware event occurs (e.g. `UART_TXDRDY` becomes true), hardware executes an interrupt handler (e.g. `uart_handler`). The handler should check that it was called for the right reason - e.g. `uart_handler` should check `UART_TXDRDY` because it might be triggered for e.g. receiving events.

Remember that, in C, any variales shared between interrupt handlers and other code should be marked as `volatile`.

Definition 7.1: An interrupt halts normal execution and executes a handler, before going back to normal execution.

Interrupts can be enabled/disabled via `intr_enable()`/`int_disable()`. This is useful for our UART example using a circular buffer, because we don't want to be interrupted while inserting a character into the buffer. Then interrupts will wait until they are enabled to be handled.

In general, interrupts might disrupt a datatype invariant.

To set up interrupts:

```
void serial_init(void) {
  ...
  UART_INTENSET = BIT(UART_INT_TXDRDY);
  enable_irq(UART_IRQ);
  txidle = 1;
}
```

The first line tells the UART interface itself that it is allowed to raise an interrupt when `UART_TXDRDY` is set to `1`. `INTENSET` stands for "interrupt enable set". This sets interrupts at the event level.

Definition 7.2: When a peripheral raises an interrupt, the NVIC (Vested Vector Interupt Controller) handles this - it is "nested" because it allows interrupts to be interrupted. The NVIC has a notion of priority to determine which interrupts can be handled where.

`enable_irq(UART_IRQ)` enables the UART interrupts in the NVIC. This sets interrupts at the peripheral level.

Calling `intr_enable()` sets interrupts at the global level.

The addresses of interrupt handlers are stored in a vectors table at the very bottom of the address space.

Remark: Interrupt handlers look like regular subroutines, inserted at arbitrary points. This means they should follow the subroutine contract, and the interrupt mechanism should allow this contract to be upheld.

The interrupt mechanism should restore all values of all registers; in particular, the interrupt handler could overwrite `r0-r3`, `psr`, `pc`, `lr` and `r12`, while still upholding the subroutine contract.

This is done using the stack.

The `lr` can't just be set to the next `pc`, because otherwise returning from the handler would just return normally; but we also need the hardware to restore the state from the stack after return. Therefore, `lr` is set to a magic value that triggers this process when `bx lr` is called.

Remark: Some advantages of this design is that interrupt handlers can be regular subroutines, and there is no need for special adapters for interrupt handlers (e.g. for storing state).

However, the interrupt latency is fixed and large.

Example (Timers): There are multiple timer circuits that can trigger an interrupt once per unit time (e.g. 1ms), by incrementing a counter every 2^n cycles and comparing it to a limit (e.g. 1000).

```
unsigned volatile ms = 0;
void timer1_handler(void) {
  if (TIMER1_COMPARE[0]) {
    millis++;
    TIMER1_COMPARE[0] = 0;
  }
}

void delay(unsigned ms_wait) {
  unsigned goal = ms + ms_wait;
  while (ms < goal) { pause(); }
}
```

where `pause` uses a `wfe` instruction (Wait For Event) to put the CPU to sleep until any event (interrupt) occurs.

Note that `goal` or `ms` could overflow, so we should make it overflow-resistant:

```
while ((ms - start_ms) < ms_wait) { ...  }
```

8. Operating systems

At the moment, if we want to run two ‘programs’ ‘at the same time’, we can’t have control flow within the programs - their state needs to be stored at the global scope, with the main function alternating between calling each program to step one iteration. We would like to be able to write each program normally, and for some main runner to do the context switching for us.

Our list of demands is:

- be able to execute multiple programs concurrently
- programs should be able to respond to external events and communicate with each other
- each program should have its own control flow
- it should seem that each program is running simultaneously
- the programmer should not need to know when to disable interrupts in order to guarantee correct code

Here we will be looking at the micro:bian operating system, created by Mike Spivey.

Micro:bian uses message passing. Messages allow process to cooperate by exchanging messsages in a way that synchronises their behaviour, and allows us to eliminate shared variables.

Micro:bian uses cooperative multitasking. This means that each program must yield control, either voluntarily or because it cannot make any more progress on its own. When a process yields, the OS decides which process to execute next. Interrupts can still occur, and interrupt a process.

Note that a yield could happen in a subroutine, so each process needs its own subroutine stack.

Startup:

```
void init(void) {
  SERIAL = start("Serial", serial_task, 0, STACK);
  TIMER = start("Timer", timer_task, 0, STACK);
  HEART = start("Heart", heart_task, 0, STACK);
  PRIME = start("Prime", prime_task, 0, STACK);
}
```

I.E. we create a fixed number of tasks at startup, including both device drivers and program processes.

The variables `SERIAL`, `TIMER` etc are global variables which represent the process IDs of the process.

Micro:bian also creates an idle task, which puts the CPU to sleep (good for energy efficiency).

Message passing example:

```
void init(void) {
  ...
  GENPRIME = start(...)
  USEPRIME = start(...)
}
```

Example of sending messages:

```
void prime_task(int arg) {
  int n = 2;
  message m;
  while (1) {
    if (prime(n)) {
      m.int1 = n;
      send(USEPRIME, PRIME, &m);
    }
    n++;
  }
}
```

Receiving messages:

```
void summary_task(int arg) {
  int count = 0;
  int limit = arg;
  message m;
  while (1) {
    receive(PRIME, &m);
    while (m.int1 >= limit) {
      printf("There are %d primes less than %d\n", count, limit);
      limit += arg;
    }
    count++;
```

```
  }
}
```

The sender waits until the message is received; the receiver waits until the message is sent. This means that the OS can transfer data when both processes are frozen, so we don't need to worry about the impact of interrupts within the process.

If two processes send a message to the same process, the OS determines ordering.

Although there are more efficient concurrency constructs than message passing, message passing gives us stronger guarantees about race conditions (or, lack of).

The message format uses 16 bytes:
- message type (2 bytes/`short`)
- sender (2 bytes/`short`)
- data (3 `int`s)

8.1. Device drivers

The OS is the only thing allowed to register interrupt handlers. It will send a message to drivers when an interrupt is received. It will also disable the interrupt in the handler, with the driver having the job of reenabling that interrupt.

When initialising the driver task, we tell the OS to send us a particular interrupt by using `connect(INTR_IRQ)` where `INTR_IRQ` is e.g. `UART_IRQ`. We also `enable_irq(...)`. We also need to tell the OS that we've handled the interrupt by calling `clear_pending(INTR_IRQ)`.

In a driver task, we continually wait to receive a message, and when it is received we switch-case on the message type.

Remark: Context switching is expensive - about 20µs per context switch.

8.2. Context switching

We want to switch processes on `send()`, `receive()`, `yield()`, or an interrupt.

These temporarily pass control to the OS, which decides where to go next.

To do this, we:
- enter the OS by either a software interrupt instruction, `svc` (supervisor call), or by a normal interrupt
- save the entire processor state on the stack
- after choosing a new process, restore state to continue.

Definition 8.2.1: The supervisor call instruction, `svc`, acts mostly like a normal interrupt:
- regs `r0-r3`, `r12`, `lr`, `pc`, `psr` are saved to stack
- magic value is placed in `lr` so that hardware knows to do special return

However, there are some differences:
- hardware moves CPU to different state (traced in `CONTROL` reg), from "user" to "kernel"

- `sp` now references a different register (`msp`, "main" stack pointer, as opposed to `psp`, "program" stack pointer), so OS has its own stack
- different magic value is placed in `lr` (so we can go back to user mode)

Remark: Each process has its own `psp`.

The only info we really need to keep track of in the OS stack is the `psp` of each process, as it's easy to return to there.

Remark: The first time a process runs, it is resumed just as if returning from a system call. So, we set up a fake stack / exception frame that contains the relevant info - an argument to the main function in `r0`, the location of the main function in the `pc`, and the location of an exit stub in `lr`, in case the process returns. All the other registers can be set to anything.

The `_proc` struct in microbian is always accessed as a `typedef struct _proc *proc` because it is too large to be carrying around in registers.

```
typedef struct _proc *proc;

struct _proc {
    int pid;                  /* Process ID (equal to index) */
    char name[16];            /* Name for debugging */
    unsigned state;           /* SENDING, RECEIVING, etc. */
    unsigned *sp;             /* Saved stack pointer */
    void *stack;              /* Stack area */
    unsigned stksize;         /* Stack size (bytes) */
    int priority;             /* Priority: 0 is highest */

    proc waiting;             /* Processes waiting to send */
    int pending;              /* Whether HARDWARE message pending */
    int filter;               /* Message type accepted by receive */
    message *msgbuf;          /* Pointer to message buffer */
    proc next;                /* Next process in ready or send queue */
};
```

Possible `state`s:

- `DEAD` - process has exited / was never started
- `ACTIVE` - process is ready to run; either the current process or in the ready queue waiting to be run
- `SENDING` - waiting to send to another process. will be in the sending queue of the destination
- `RECEIVING` - waiting to receive a message
- `SENDREC` - sending and receiving
- `IDLING` - idle process (runs when no other process is ready)

Message sending rules: either:

- The destination process is `RECEIVING` and can receive message

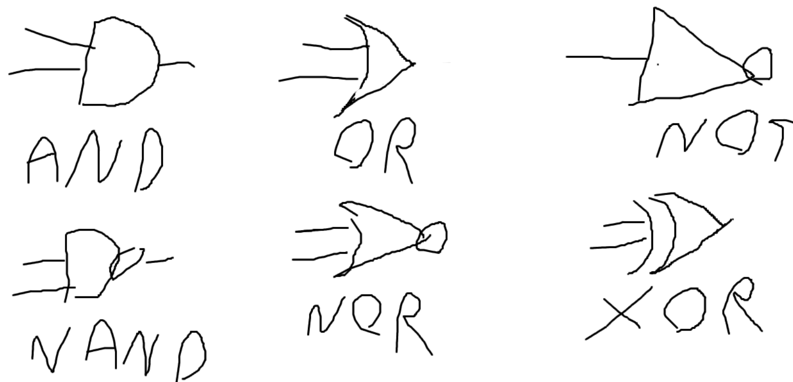
  - transfer message to destination; destination `READY`; add to queue
- Or the process cannot receive (either because it isn't `RECEIVING`, or it's receiving filter is for a different type of message)
  - Keep sender in queue until destination is ready to receive

For every process, the next process trying to send to it is stored in `waiting`; the rest of the send queue is chained through `next`. This doesn't conflict with the `ready` queue because any sending proc is not `ACTIVE`.

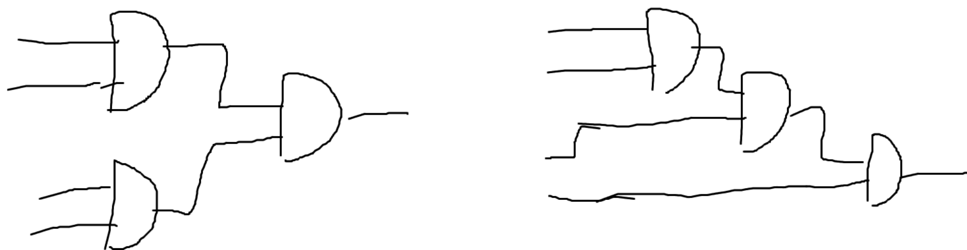
9. Combinational logic

Abstractly, the CPU and its instructions are a function mapping bitstrings to bitstrings. We would like to be able to specify these transition functions (see def Turing machine TODO).

We need mathematical foundations to specify these functions, as well as a way to physically implement such functions: logic gates.



Note that, when chaining gates, we can choose different ways to represent the same expression because of associativity:



In general, we should aim to use fewer logic gates. However, we also need to consider that it takes time for signals to settle (propagation delay); logic gates need to be evaluated sequentially, so we should aim to have fewer sequential gates.

Theorem 9.1: Any truth table has a circuit using and/or/not that implements that truth table.

Proof: For each "1" in the output, construct an expression that is "1" only for the corresponding inputs. I.e. ANDs with NOTs for zero inputs. Then take the disjunction of all of these. □

Example (MAJority voting): The following function is 1 if the majority of the inputs are 1:

a	b	c	MAJ
1	1	1	1
0	1	1	1
1	0	1	1
1	1	0	1
TODO			

For each 1, we have:

$$a \land b \land c$$
$$\neg a \land b \land c$$
$$a \land \neg b \land c$$
$$a \land b \land \neg c.$$

So overall the function is

$$(a \land b \land c) \lor (\neg a \land b \land c) \lor (a \land \neg b \land c) \lor (a \land b \land \neg c).$$

Note that this is in disjunctive normal form (see IPS).

Note also that this is not necessarily an optimal circuit.

Example: The MAJ function can also be represented as

$$(a \land b) \lor (a \land c) \lor (b \land c).$$

Remark: Methods for finding minimal "sum of products" for better circuits exist, e.g. Karnaugh maps, but are beyond the scope of this course.

Example (Multiplexer): $z = c\,?\,b : a$.

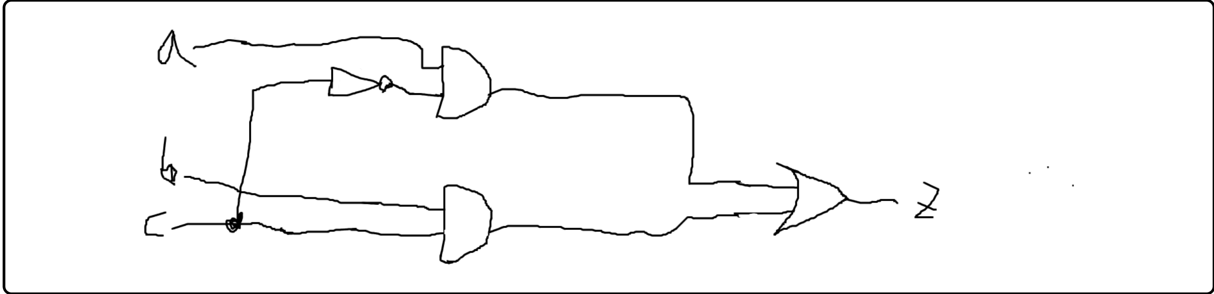
Truth table: TODO

Naïvely we might determine the formula to be

$$(a \land b \land c) \lor (\neg a \land b \land c) \lor (a \land b \land \neg c) \lor (a \land \neg b \land \neg c).$$

The first two, and last two, terms can be simplified:

$$(b \land c) \lor (a \land \neg c).$$



TODO xor. two options.

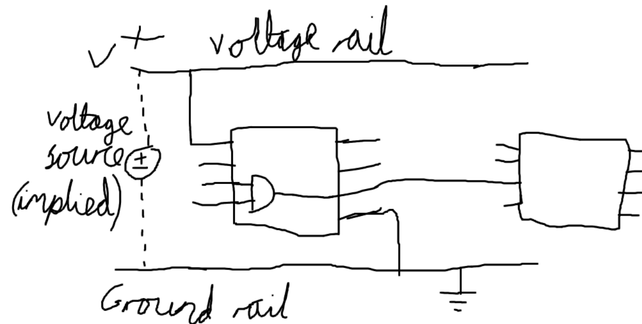
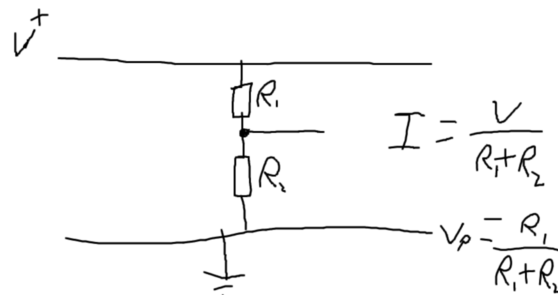


Figure 7: Rough high-level overview of what a logic gate looks like. Here we have an output chained to another logic gate.

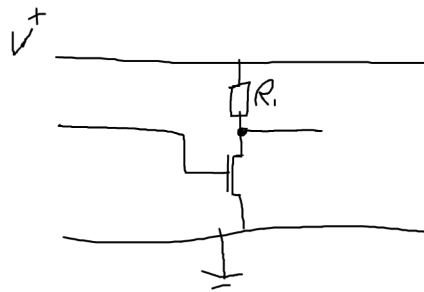
Definition 9.2: TODO Kirchoff's current law



TODO should that be R2 / ...?

Here V_p is the output voltage.

If we wanted to implement a NOT gate, we might want a high voltage to give a low resistance for R_2, and vice versa. It turns out that transistors can do this for us.



Definition 9.3: n-type MOSFET TODO what does this stand for?

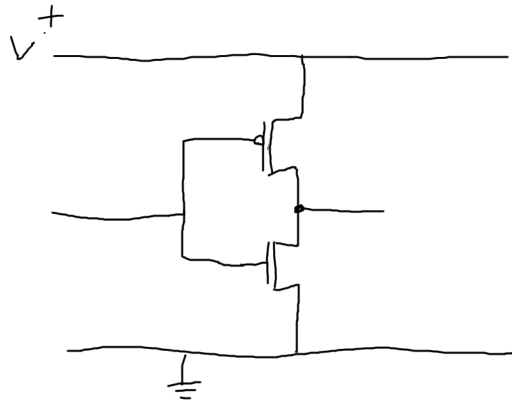
TODO circuit-ish diagram

TODO I-V characteristics

Remark: The biggest bottleneck to running CPUs faster is heat dissipation. The circuit shown above has this problem when TODO.

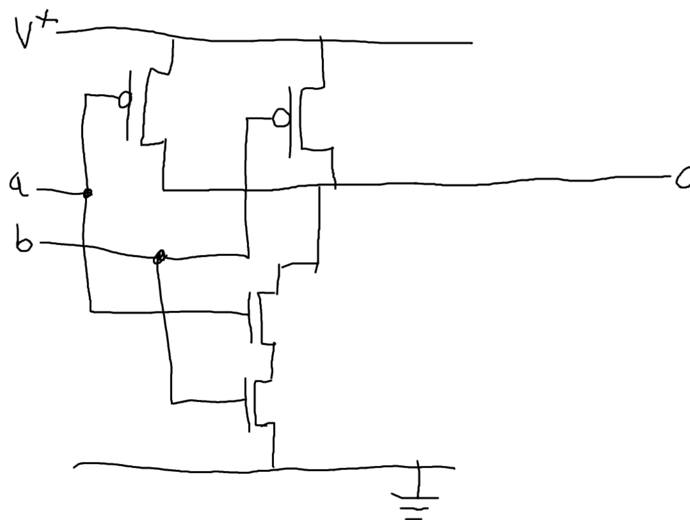
TODO p-type MOSFET

TODO CMOS



TODO pull-up and pull-down

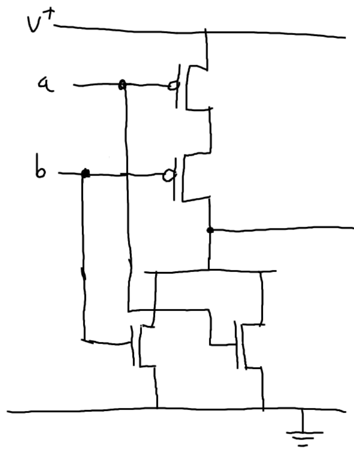
Example: This is a NAND gate:



TODO why?

TODO complementary circuit. what is it? why? Change series p-type to parallel n-type, etc.

Example: This is a NOR gate:



TODO multi-input (N)AND?

TODO Why choose to chain gates rather than do it in one?

10. Sequential circuits

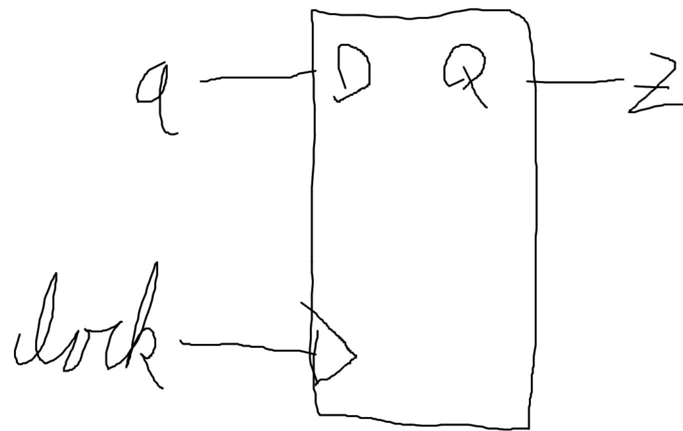
Definition 10.1: A sequential circuit is a circuit that depends on the history of its inputs.

TODO clock signal graph

Definition 10.2: A synchronous circuit is one where:
- A clock signal is shared between all components
- Circuit changes state on rising edge
- Signals must be stable for a given setup time, before the next rising edge

Remark: In reality it might be useful to turn the clock off for some components to avoid unneeded heat dissipation.

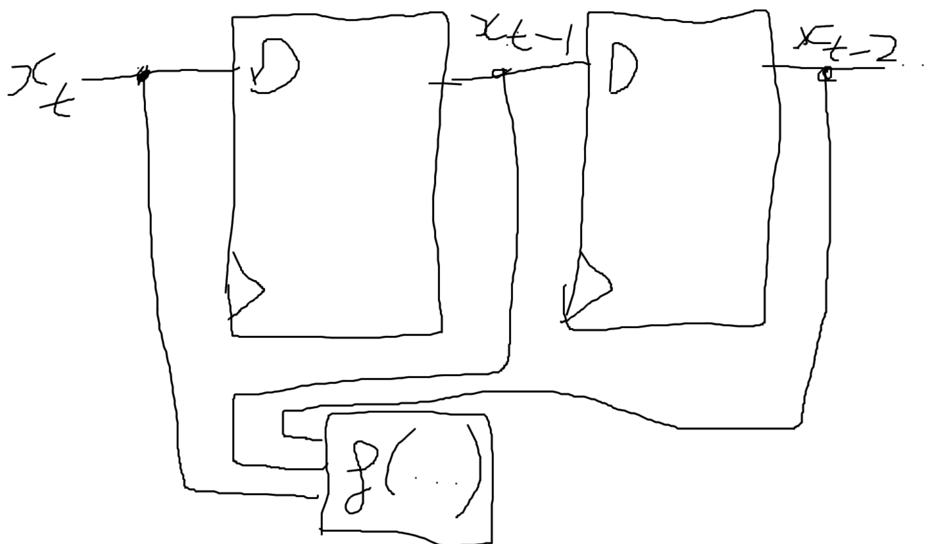
Definition 10.3: The D-type flip-flop:



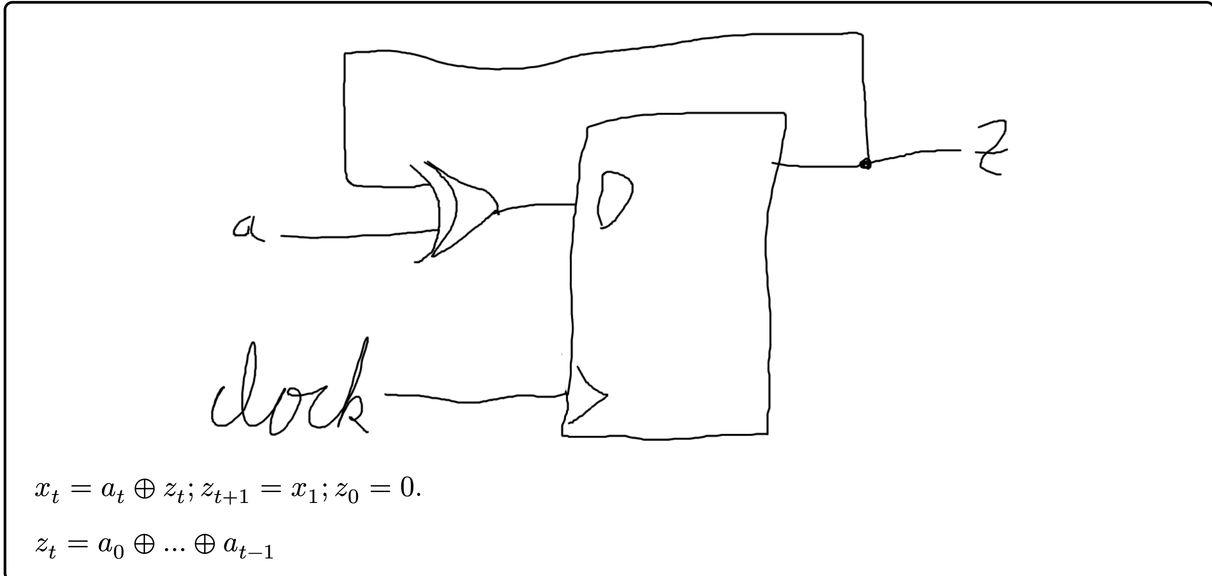
a_t	z_t	z_{t+1}
0	0	0
0	1	0
1	0	1
1	1	1

The D-type flip-flop essentially acts as a delay from a to z.

Definition 10.4: A shift register allows us to compute a function from a fixed finite number of previous states:



Example: An example of using arbitrary previous state is parity detection, a circuit that outputs 0 if even number of 1s in the inputs $a_1, ..., a_t$, 1 if odd number of 1s in the a_is.



$x_t = a_t \oplus z_t; z_{t+1} = x_1; z_0 = 0.$

$z_t = a_0 \oplus ... \oplus a_{t-1}$

Example (Pulse shaper): Takes a possibly irregular input pulse and outputs a pulse that is 1 cycle long.

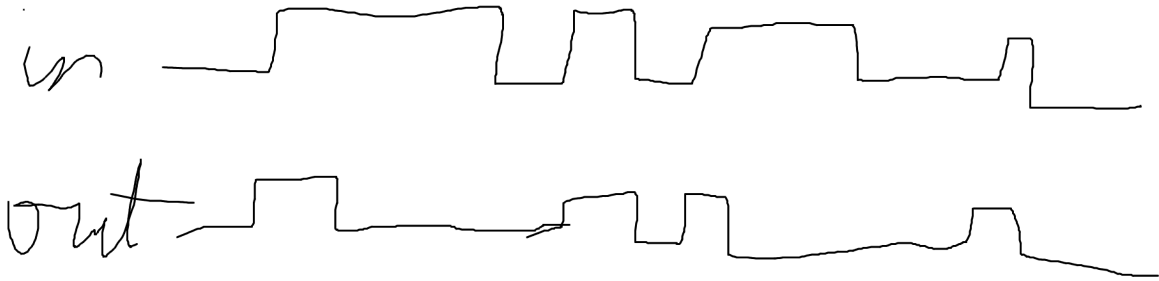


Figure 16: Example input and output

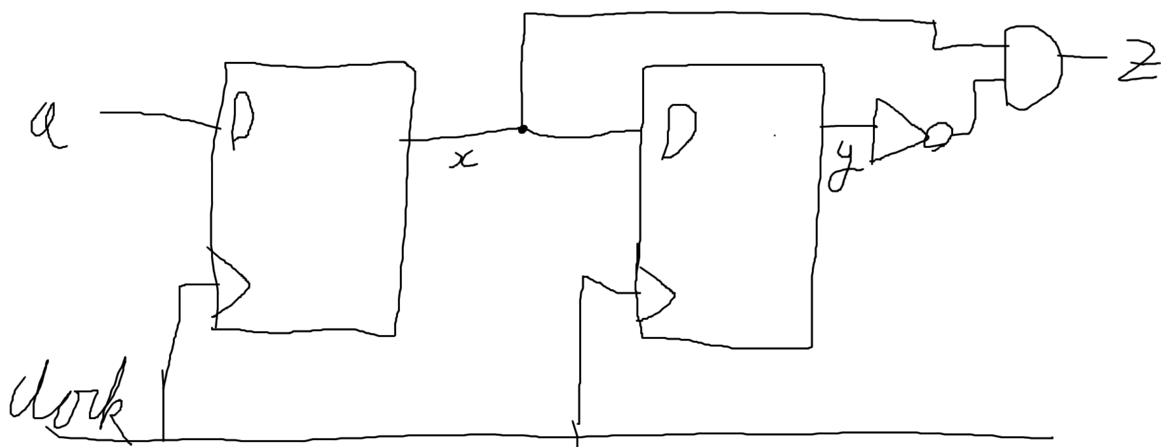


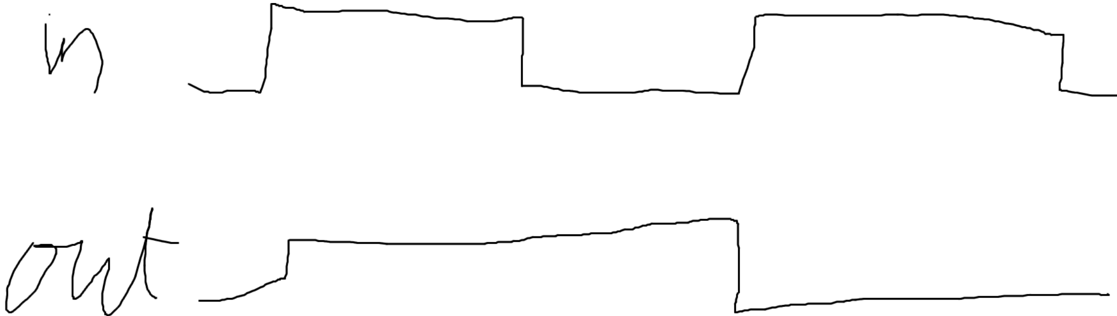
Figure 17: Circuit diagram for pulse shaper

We need the first flip-flop to make sure that the output pulse is exactly one cycle long.

Definition 10.5: In Mealy machines, output depends on state and inputs.

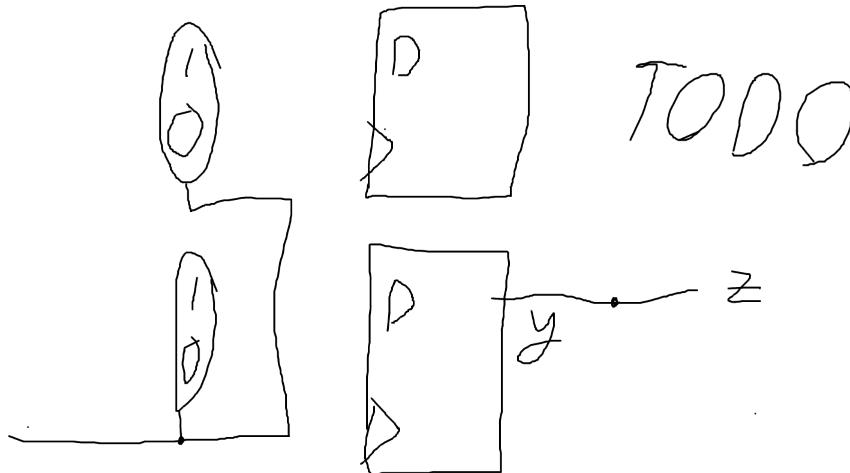
In Moore machines, TODO

Example (Bathroom light switch):



Pseudocode (input `a`, output `y`, additional state `x`, `y`)

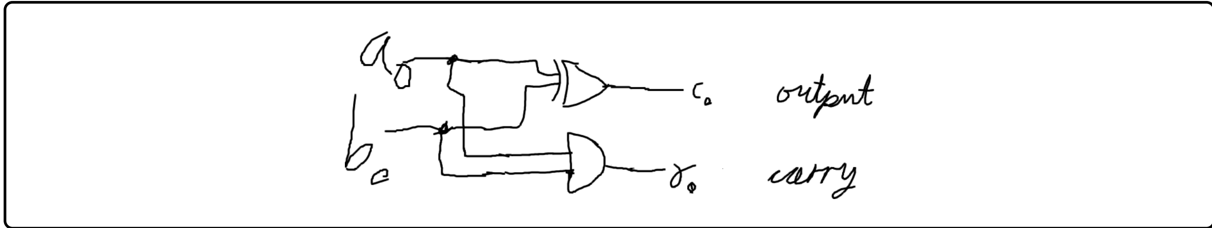
```
forever {
  z = y;
  pause;
  x = (a ? x : y);
  y = (a ? !x : y);
}
```



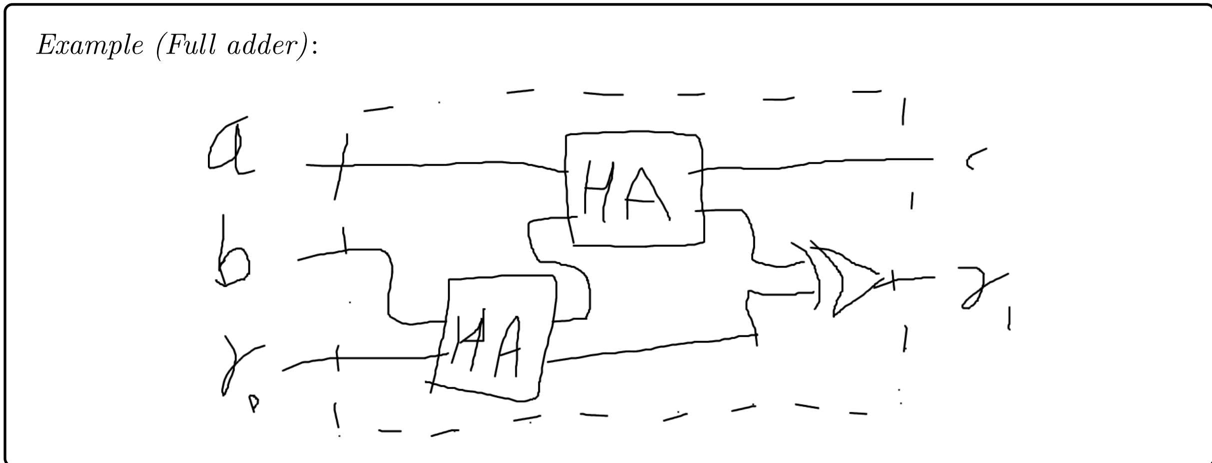
11. Architectural elements

When specifying complicated circuits, we should abstract things away into smaller "functions".

Example (Half adder):



Example (Full adder):

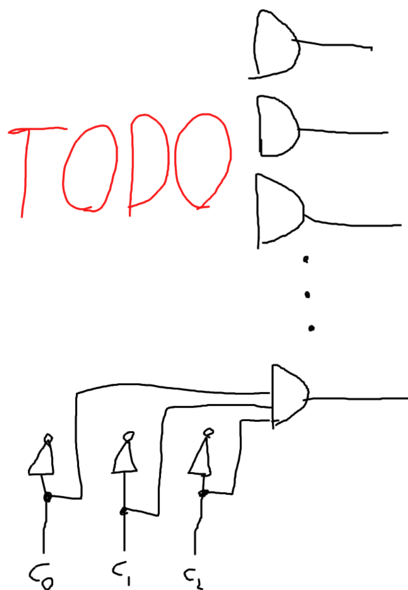


TODO ripple carry adder

TODO carry lookahead adder

TODO one-hot encoding, multi-input multiplexer. Don't need to check for other inputs 0 because assumed that only 1 input high.

We want to take a binary encoding in n bits and turn it into a one-hot encoding, with 2^n outputs, to put it into a multi-input multiplexer.



TODO n-way multiplexer

We represent a collection of wires by a wire with a line through it, with a number representing how many bits we're representing.

We will represent a collection of D-type flip-flops by having a collection of wires in and out of a single flip-flop circuit symbol.

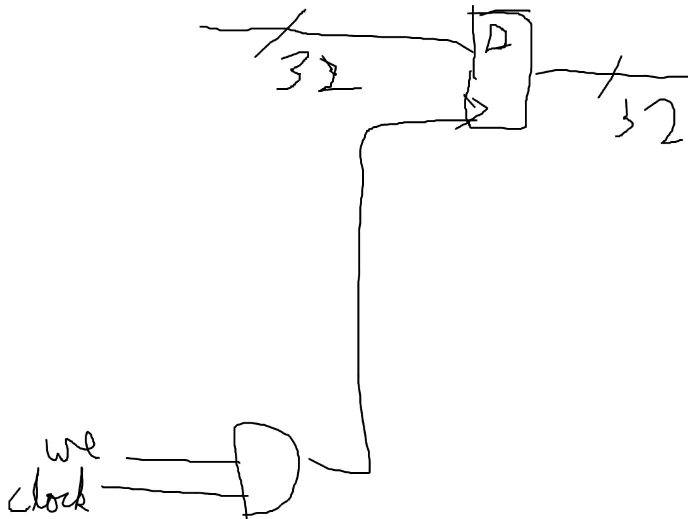
TODO addition/subtraction circuit

TODO ROM

> **Remark**: We could replace any circuit by ROM. But this isn't really a good idea because it's near-impossible to actually specify the output for every single input.

TODO registers

Write-enable registers:



But this can cause problems with clock signal propagation delay.

TODO write-enable via multiplexing with input and previous state

TODO register files; allows us to select which register to write to, and also pass a write-enable flag. We need a separate write-enable flag because not all instructions write to a register. `rd1` and `rd2` tell us which two registers we would like to read.

We also need to consider how to manage special registers: `pc` should always be written to, and should always be 4 higher than the current instruction address; and `lr` should sometimes be written to with the next value of the `pc`. TODO

We can implement constants to add to by connecting wires directly to the high voltage or ground rails.

11.1. ALU

TODO symbol

Take as input two 32-bit bundles of wires for two operands, and control signal. Outputs result of numerical calculation, and status bits.

TODO barrel shifter

To do a right-shift using a left-shift circuit, we just reverse the order of the bits at the input and output.

12. Building a datapath

Definition 12.1:

Definition 12.2: decoded signals

depend on opcode only.

Definition 12.3: derived signals

depend also on variable fields

Definition 12.4: dynamic signals depend on state of CPU

Index

Addressing modes 6
Anode 8
Baud rate 10
Cathode 8
Computer 2
D-type flip-flop 21
General Purpose Input Output (GPIO) . 7
Interrupt 12
LED multiplexing 8
Little endian 3
Mealy machines 23
Moore machines 24
NVIC 12
RAM 3
RISC 2
ROM 3
Sequential circuit 21
Serial contract 10
Setup time 21
Shift register 22
Subroutine contract 5
Supervisor call 15
Synchronous circuit 21
Turing machine 2
Two's complement 5
Universal Asynchronous Receive Transmit (UART) 10
Von Neumann Architecture 2