CSM CS61C Note #4: Compiler, Assembler, Linker, Loader (CALL), Intro to Synchronous Digital Systems (SDS)

Anthony Han

1 Compiler, Assembler, Linker, Loader (CALL)

Compiler

- **Maps high-level language code** (e.g. foo.c) **to assembly language code** (e.g. foo.s for RISC-V), matching the *calling convention* for the architecture.
- Output may contain pseudo-instructions.
- Steps in the Compiler:
  - **Lexer** - turns input into tokens, recognizes problems in tokens.
  - **Parser** - turns tokens into an **"Abstract Syntax Tree"**, recognizes problems in program structure.
  - **Semantic Analysis and Optimization** - checks for semantic errors, may reorganize code to make it better.
  - **Code Generation** - output assembly code.

Assembler: A Dumb Compiler for Assembly Language

- **Maps assembly language code** (e.g. foo.s) **to object code, information tables** (e.g. foo.o).
  - Reads and uses **directives**.
  - Replaces pseudo-instructions.
  - Produces **machine language** rather than just assembly language.
  - Creates **object file**.
- **Assembler Directives** - give directions to assembler, but do not produce machine instructions.
  - **.text:** subsequent items put in user text segment (machine code).
  - **.data:** subsequent items put in user data segment (binary representation of data in source file).
  - **.globl sym** - declares *sym* global and can be referenced from other files.
  - **.string str** - store the string *str* in memory and null-terminate it.
  - **.word w1, ..., wn** - store n 32b quantities in successive memory words.

- **Pseudo-Instruction Replacement** - assembler treats convenient variations of machine language instructions as if real instructions.

Pseudo	Real
`nop`	`addi x0, x0, 0`
`not rd, rs`	`xori rd, rs, -1`
`beqz rs, offset`	`beq rs, x0, offset`
`bgt rs, rt, offset`	`blt rt, rs, offset`
`j offset`	`jal x0, offset`
`ret`	`jalr x0, x1, offset`
`call offset` (if too big for just a `jal`)	`auipc x6, offset[31:12]` `jalr x1, x6, offset[11:0]`
`tail offset` (if too far for a `j`)	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`

- **Producing Machine Language**
  - For simpler cases like arithmetic and logical operations, we can easily convert them into binary representation.
  - For **PC-relative** branches (or jumps) within the same file, once the pseudo-instructions are replaced, we would know the number of instructions to branch. When jumping to another file, however, the offset cannot be determined at the assembler stage.
  - The assembler needs to **take two passes over the program** due to the **"forward reference problem"** – instructions can refer to labels that are forward in the program. The first pass remembers position of labels as we expand pseudo-instructions, while the second pass uses label positions to generate code.

```
1 L1: beq t0, x0, L2
2     ...
3 L2: add t1, a0, a1
```

  - Reference to static data by la requires the full 32b address of that data. We need to **store this information in the symbol table and the relocation table.**

- **Symbol Table** - list of "items" in this file that may be used by other files.
  - **Labels** - function calling.
  - **Data** - things in the .data section; variables that **may be accessed across files.**

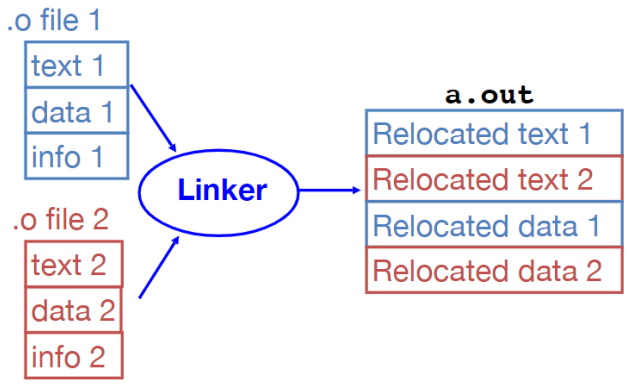
- **Relocation Table** - list of "items" this file needs the address of later.
  - Any **external label** jumped to with jal.
  - Any piece of **data in static section.**

- **Object File Format**
  - **Object file header** - size and position of the other pieces of the object file.
  - **Text segment** - the machine code.
  - **Data segment** - binary representation of the static data in the source file.

  - **Relocation information** - identifies lines of code that need to be fixed up later.
  - **Symbol table** - list of this file's labels and static data that can be referenced.
  - **Debugging information** - mapping from machine code to high-level language code; a standard format is ELF, except Microsoft.

Linker

- **Maps object code files and information tables** (e.g. foo.o, libc.o) **to executable** (e.g. a.out).
- Combines several object (.o) files into a single executable ("linking").
- **Enables separate compilation of files:**
  - **Changing one file do not require recompilation of entire program.**
  - Old name "link editor" as it edits the "links" in jump and link instructions.



- Static linking process:
  - Step 1: Take text segment from each .o file and put them together.
  - Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
  - Step 3: **Resolve references** - go through relocation table; **fill in all addresses.**
    - Relocation editing is needed for **external jal**. However, note that PC-relative addressing is preserved even if code moves, so **conditional branches do NOT need relocation editing.**

      ```
      1 || jal ra, printf
      ```

    - To resolve references:
      - **Search** for reference **in all user symbol tables.**
      - If not found, **search library files** (e.g. for printf).
      - Once absolute address is determined, fill in machine code appropriately.

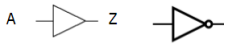
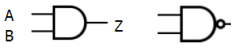
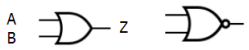
Loader

- **Loads the executable** (e.g. a.out) **from the disk into memory and runs it.**
- **In reality, loader is the operating system.**
  - Loading is one of the OS tasks.
  - Nowadays, the loader does a lot of the linking as the linker's "executable" is actually only partially linked, still having external references.
- Loading process:
  - Reads executable file's header to determine size of text and data segments.
  - Creates new address space for program large enough to hold text and data segments, along with a stack segment.
  - Copies instructions and data from executable file into the new address space.
  - Copies arguments passed to the program onto the stack.
  - Initializes machine registers - most registers cleared, stack pointer assigned address of first free stack location.
  - Jumps to start-up routine that copies program's arguments from stack to registers and sets the PC. If main routine returns, start-up routine terminates program with the exit system call.

2 Intro to Synchronous Digital Systems (SDS)

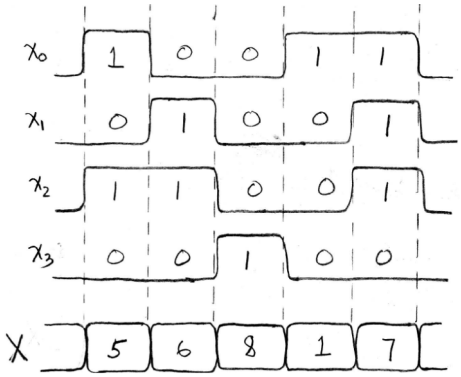
Combinational Logic Symbols

Common combinational logic systems have standard symbols called logic gates:

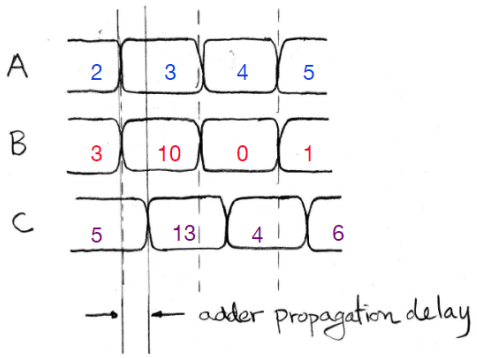
- Buffer, NOT 
- AND, NAND 
- OR, NOR 

Signals and Waveforms

Signals and waveforms of digital circuits **show time and grouping.** For a digital signal $x = \overline{x_3 x_2 x_1 x_0}$, we can have waveform



Since it is a circuit, there will be **circuit delay.** Consider a 4b adder $c = a + b$:



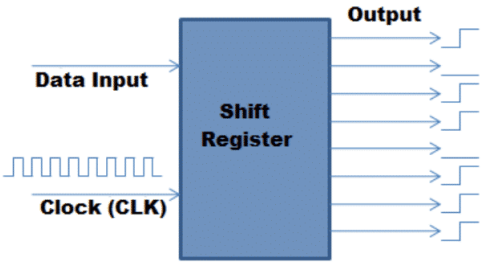
Type of Circuits

Synchronous Digital Systems consist of two basic types of circuits:

- **Combinational Logic (CL)** circuits.
  - Output is **a function of inputs only**, not the history of its execution.
  - E.g., circuits to add A, B (ALUs).
- **Sequential Logic (SL)** circuits.
  - Circuits that **"remember" or store information**, aka **"State Elements."**
  - E.g., memories and registers (Registers).

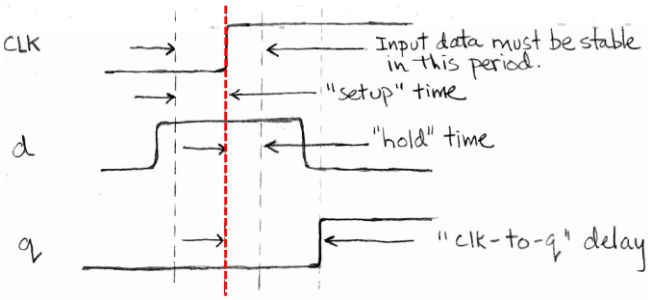
Register Basics

Due to the implementation of a register, the input D is sampled and transferred to the output Q **only on the rising edge of the clock**; at other times, the input is ignored.

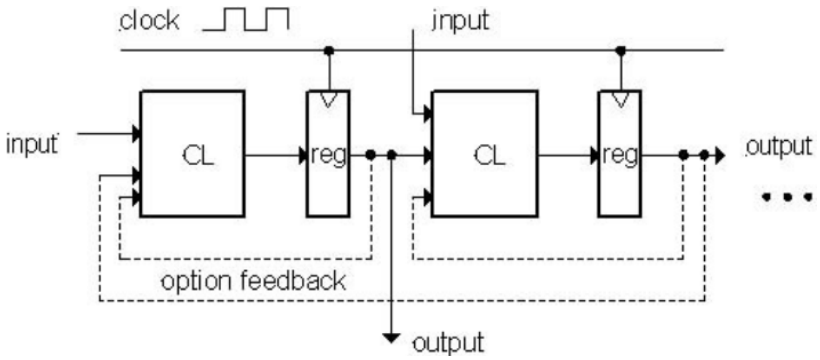


To make sure that the output is consistent, we define the following terms:

- **Clock (CLK)** - steady square wave that synchronizes system.
- **Setup Time** - when the input must be stable *before* the edge of the CLK.
- **Hold Time** - when the input must be stable *after* the edge of the CLK.
- **"CLK-to-Q" Delay** - how long it takes the output to change, measured from the edge of the CLK.



Model for Synchronous Systems

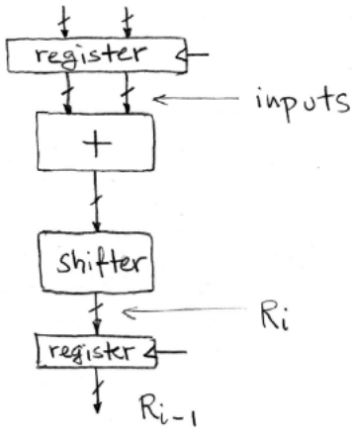


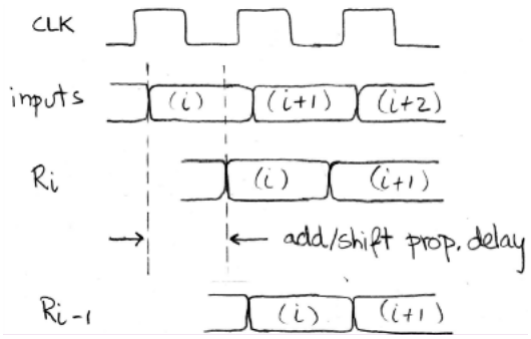
Note that the maximum clock frequency is achieved when the period is minimal.

- **The clock period must be longer than the critical (worst-case) path.**
  - Otherwise, the output is updated before the desired result arrives.
  - To compute the critical path, **add:**
    - **Clk to Q time** - necessary to get the output of the registers.
    - ***Worst case* combinational logic delay.**
    - **Setup time** for the next register.
- **The Clk to Q time plus *best case* combinational logic delay must be more than hold time.**
  - Otherwise, you don't hold the input to the register long enough, hence **losing information**.
  - **Solution: add delay on best-case path** (e.g. two inverters).

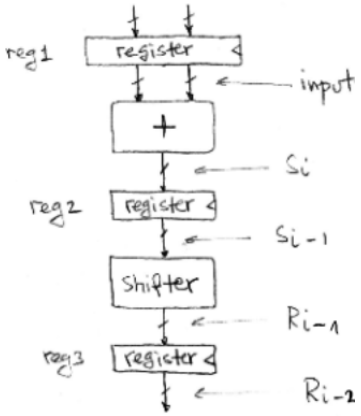
Pipelining to Improve Performance

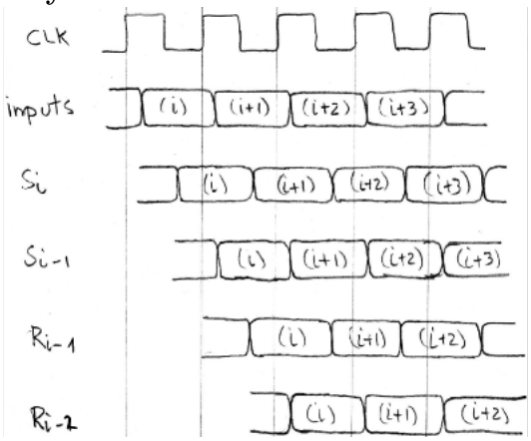
If we try to do multiple things between two registers, our **clock period must be larger than the propagation delay of the combinational logic components combined**; thus yielding a low frequency.





We can use **pipelining to increase the frequency of the circuit:**





- **Adding an extra register** yields **higher clock frequency.**
- **More outputs per second** - higher bandwidth.
- Each individual result takes longer - **greater latency.**