Tricks de FDS (part II)

1. FSM en Verilog

- next-state logic (en always @*).
- next-state update (en always @(posedge clock) avec gestion du reset).
- output update (en always @*).

2. Trouver la fréquence maximale

- calculer tous les temps $t_{\text{setup}} + t_{\text{comb}_{\text{max}}} + t_{\text{cQ}_{\text{max}}}$ pour chaque chemin entre deux flip-flops.
- prendre le plus grand temps.
- calculer la fréquence maximale : $f_{\text{max}} = \frac{1}{t_{\text{max}}}$.

3. Vérifier les hold violations

- calculer tous les temps $t_{\text{comb}_{\text{min}}} + t_{\text{cQ}_{\text{min}}}$ pour chaque chemin entre deux flip-flops
- comparer avec t_{hold}

4. Cours

- set-reset latch: type de circuit de mémoire basique utilisé en électronique. deux entrées : S et R. Elles sont asynchrones (pas besoin que la clock passe en front montant ou en front descendant pour qu'elles marchent) ! Elles ne dépendent que des inputs.
- D latch : comme un set-reset latch, mais plus contrôlable (input R devient input C qui nous permet de contrôler quand on veut que l'output du latch se verrouille ou suive la valeur de S).
- D flip-flops : comme un D latch, mais l'output ne prend la valeur de S que quand l'input C change de valeur (devient vrai). Quand il est vrai, même si S change, l'output reste fixe (à la valeur qu'il avait quand C était en front montant).

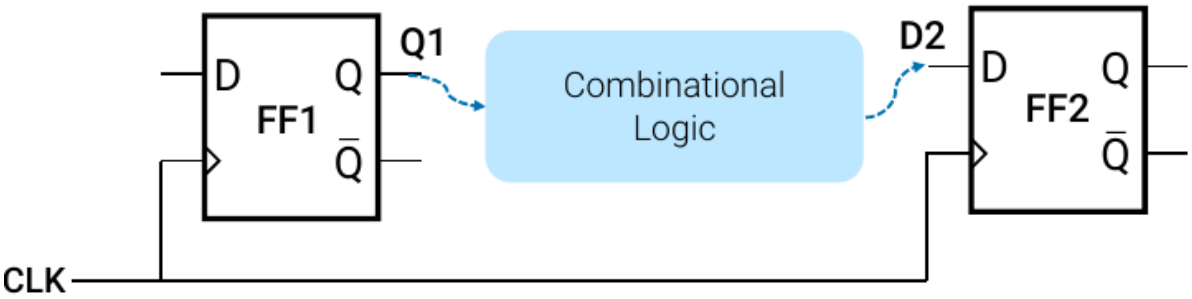
→ on utilise l'horloge pour décider quand notre state change, pour des soucis de synchronisation ! En effet si les composants commencent à maj leur valeur n'importe quand, et d'autres à un moment précis, on a des pb de stabilité (pendant un court moment, un output peut être invalide).

- FSM (finite state machine) –> if Moore then no input (it only takes the previous state), if Mealy it takes both the previous input **and** other inputs
- clock duty ratio : $\frac{\text{time of high sign}}{\text{duration of low sign}}$

5. Setup and Hold times

Contamination Clock To Q: quand la clock se déclenche, le temps minimum que peut mettre Q à changer.

Propagation Delay Clock to Q: quand la clock se déclenche, le temps maximum que met Q à se stabiliser.



Pour que D2 prenne correctement la valeur (pas de métastabilité), il faut que Q soit stable, au moins

- t_{setup} : avant que la clock ne se déclenche
- t_{hold} : après que la clock se soit déclenchée

Donc on veut check :

- $t_{\text{cQ}_{\text{max}}} + t_{\text{comb}_{\text{max}}} + t_{\text{setup}} \leq t_{\text{clock}}$ (que le temps avant que les calculs commencent + les calculs + le temps de setup soient inférieurs au temps de la clock)
- $t_{\text{cQ}_{\text{min}}} + t_{\text{comb}_{\text{min}}} \geq t_{\text{hold}}$ (que le temps avant que les calculs commencent + les calculs soient supérieurs au temps de hold sinon la valeur n'est pas tenue)

Valerio se tourne, va chercher un stylo chez Habib, et se retourne vers moi. Je prends le stylo toutes les 5 secondes → **clock !** et :

- j'ai besoin de voir le stylo 2s avant de commencer à le prendre (le moment où je vais arriver à 0 mod 5s) → **t_setup !** Cette condition est violée si je vais trop vite (la clock est trop rapide), que Valerio a pas le temps de me montrer le stylo. Soit Valerio doit aller plus vite (on diminue le **t_comb**), soit on doit me ralentir (on diminue le **clock**).
- je mets 1s à prendre le stylo **t_hold !** Cette condition est violée si Valerio va chercher un autre stylo chez Habib trop vite.

$$\text{on veut que la valeur soit bonne avant le } t_{\text{setup}}$$
$$\Leftrightarrow t_c Q + t_{\text{comb}} \leq t_{\text{clock}} - t_{\text{setup}}$$

Maintenant on rajoute un t_{skew}, qui décale un peu le t_{clock} (c'est du temps bonus qu'on ajoute) :

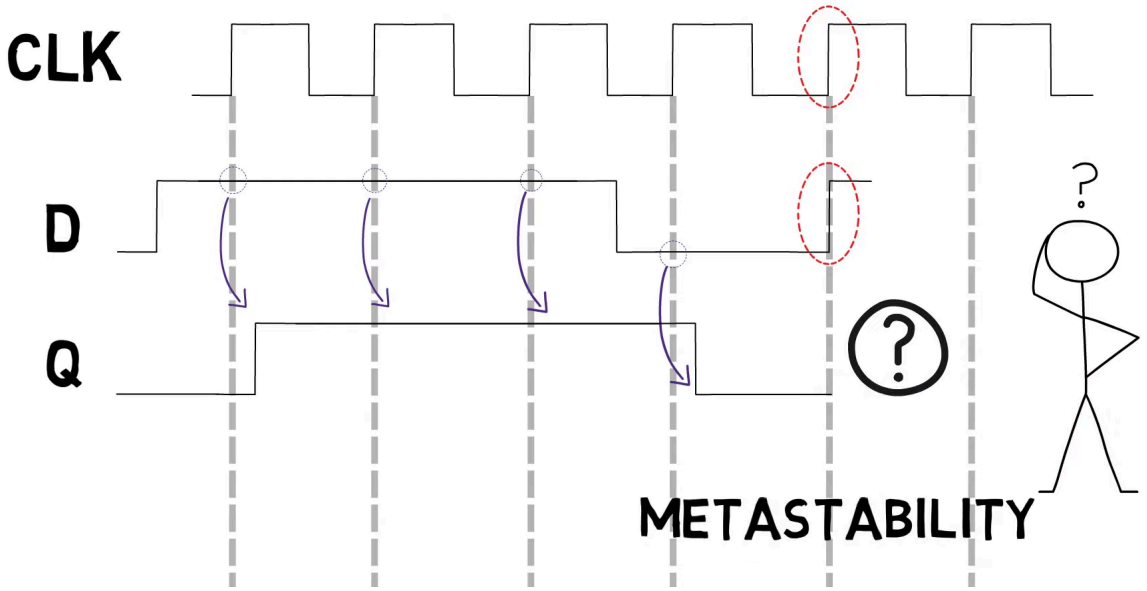
$$\Leftrightarrow t_c Q + t_{\text{comb}} \leq t_{\text{clock}} - t_{\text{setup}} + t_{\text{skew}}$$

Calculer le temps entre FF A vers FF B pour calculer f_{max}

$$t_c Q_{\text{max}} + t_{\text{comb}} + t_{\text{setup}} + t_{\text{skew\_du\_FF\_A}} - t_{\text{skew\_du\_FF\_B}} = \text{delay}$$

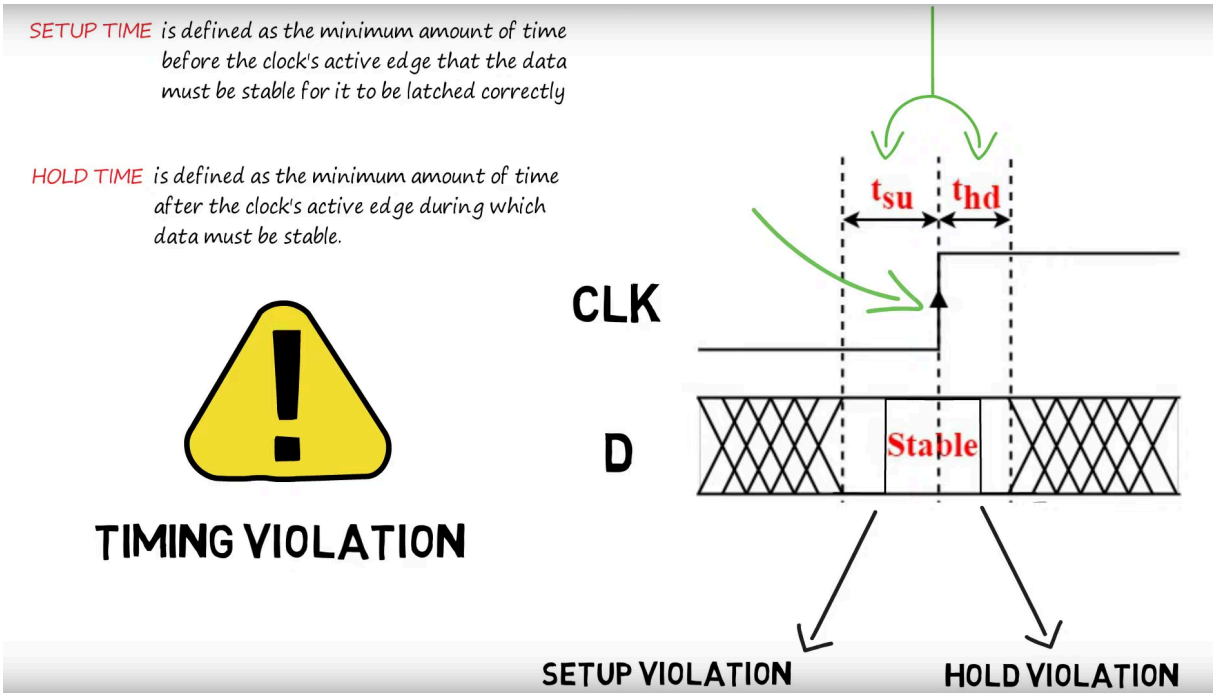
Vérifier le hold:

$$t_c Q_{\min} + t_{\text{comb}} + t_{\text{skew\_du\_FF\_A}} - t_{\text{skew\_du\_FF\_B}} = \text{delay}$$



solution : mettre deux flip flop en série. si le premier est métastable, le signal va se stabiliser entre les deux et le deuxième flip flop va prendre un signal non métastable.

https://www.youtube.com/watch?v=xCA54Qu4WtQ

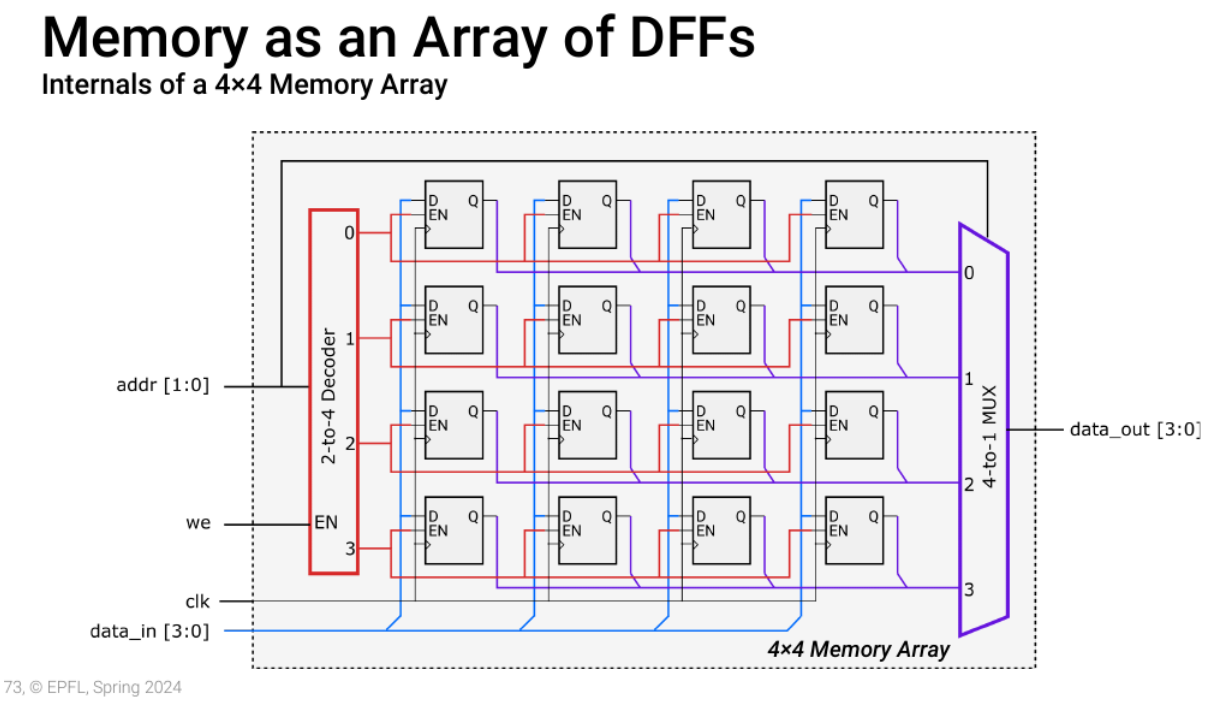


6. Clock Skew

différence de temps entre le moment où la clock se déclenche pour le flip flop 1 et le flip flop 2. si le skew est positif, c'est cool ! on a plus de temps pour faire les calculs. si le skew est négatif, c'est moins cool, on a moins de temps pour faire les calculs.

7. Memory

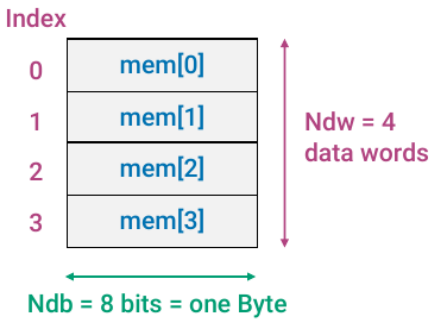
Il y a deux MUX : un pour activer l'écriture au bon endroit (qui ne s'active que quand `WE` est vrai), un pour choisir ce qui sort.



7.1. Memory dans Verilog

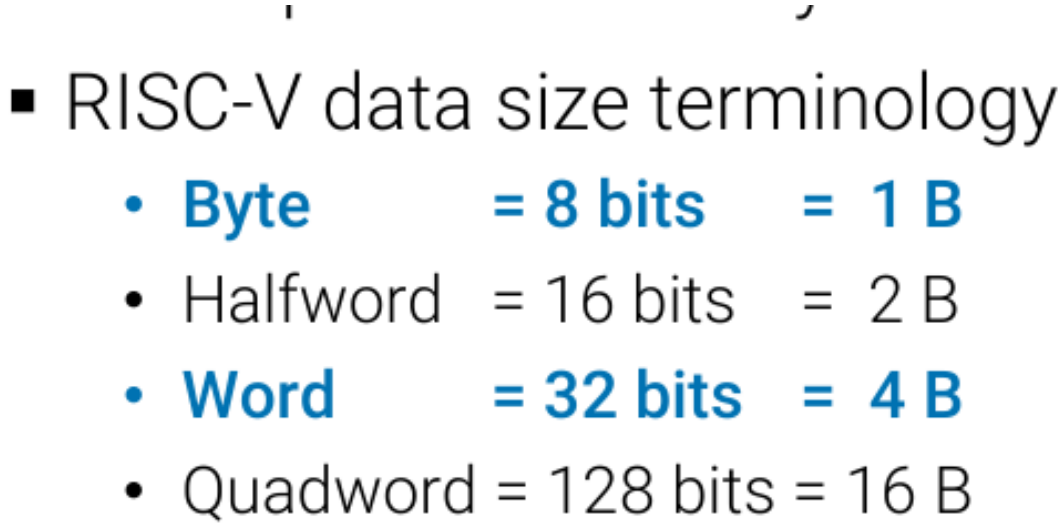
```
reg [nombre-de-bits-par-word-1:0] nom-de-la-variable [nombre-de-words-1:0];
```

```
reg [Ndb-1:0] mem [Ndw-1:0];
```



8. RISC-V

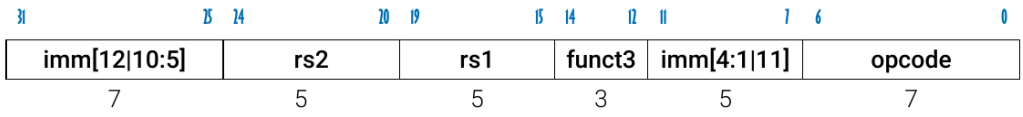
- `li [reg] [value]`: load an immediate into a register
- `and [reg1] [reg2] [reg3]`: bitwise and, stores the result in reg1
- `add [reg1] [reg2] [reg3]`: add, stores the result in reg1
- `srai [reg1] [reg2] [value]`: shift right “arithmetic” (there is a “logic” version) immediate, stores the result in reg1
- `bne [reg1] [reg2] [label]`: branch not equal, jumps to label if reg1 and reg2 are not equal



8.1. Instruction de type Branch

On stocke un `offset` décalage `immediate` qui dit de cb d'instructions on veut sauter, c'est toujours un multiple de 2 (et souvent même de 4 car on ne veut pas sauter au milieu d'une instruction). C'est pour ça que nous n'avons pas besoin de stocker le dernier bit de `immediate` (qui est toujours 0). Il peut donc faire 13 bits (car on peut stocker 12 bits et on sait que le dernier est 0).

Attention il est stocké de façon wtf. On stocke ensemble les bits de 4 à 1, le bit 11, et ensuite le reste.



- **rs1:** 5-bit index of the first operand register
- **rs2:** 5-bit index of the second operand register
- **opcode:** $(1100011)_2$, 7-bit operation code specifying the operation type (branch)
- **funct3:** 3-bit function code for branch condition (BEQ/BNE/BLT/BGE/BLTU/BGEU)
- **imm:** 12-bit immediate, also called **offset**, which encodes a signed offset in multiples of two bytes. Note that imm[0] is not needed (because of the multiplication by two), which is why it is replaced by imm[11]. This arrangement keeps imm[4:1] and imm[10:5] in their usual place, which simplifies hardware implementation.

8.2. Instruction de type Memory

- `lw t1, 12(t2)` : load word, va chercher la valeur à l'adresse `12 + t2` et la met dans `t1`

8.3. Instruction de type Jump (non conditionnels)

comme les instructions de type branch mais utilisés pour sauter qqpart sans condition (comme lors d'un appel de fonction)

- `jal return_address, label/immediate` - jump and link, va à `label` et stocke l'adresse de retour (c'est à dire l'adresse de l'instruction juste après le jump (donc pc + 4)) dans `return_address`. Attention quand on précise un `immediate`, il sera ensuite multiplié par 2 (car on ne peut pas sauter au milieu d'une instruction).
- `jalr return_address, t1, immediate_offset` - jump and link register, va à l'adresse stockée dans `t1` et stocke l'adresse de retour dans `return_address`... par contre ici le `immediate_offset` n'est pas multiplié par 2.

8.4. Données

```
.data
  matrix:
    .byte 12, 78, 35, 11, 34, 113, 46, 122, 56, 24, 57, 33
```

```
.text
  # la is a pseudoinstruction that the assembler converts
  # into two lower-level instructions: auipc x8 ADDRESS_OF_MATRIX
  # and addi x8 x8 -(INITIAL_PC) (register s0 is an alias for x8).
  la s0, matrix
  addi s1, zero, 4
  addi s2, zero, 3
  add s3, zero, zero

  start:
```

voir série papier.