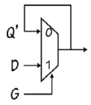
Week 3 – Seq Logic

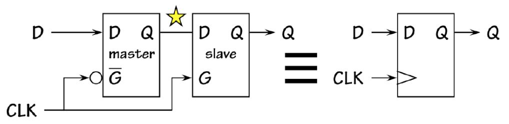
D-Latch: mux w feedback loop.



G=1, "write" D->Q indpt of Q'
G=0, "read memory "
Q follows Q' indpt of D

Dynamic Discipline

- Addresses problem of storing invalid information (if G△es 1 to 0 when D is △ing (invalid), new invalid info will be stored on D)
- Tsetup = the min time that the voltage on wire D needs to be stable before the clock edge △es from 1 to 0. = 2Tpd
- Thold = min time that the voltage on wire D needs to be stable after the clock edge △es from 1 to 0. = Tpd
- Tpd = propagation delay of D-latch (mux)



Flip-Flop

- Addresses problem of unstable O/P cos of unstable INP (inp unstable, if G=0, all inp from D will be passed to Q regardless of whether inp is valid/stable)
- When CLK=0, G of master receives 1 (inv) and G of slave receives 0 => master="write", slave="read".
- Only one D-latch is on "write" at a time.
- Output wire Q from slave changes only when the CLK rises from 0 to 1
- Tcd of a register (seq logic w reg & CLs)= time taken for invalid CLK input to invalid output on D (final output of seq logic unit).
Tpdreg = valid > valid
- DD for flip-flop: **Tcdmaster > Tholdslave**
(★ needs to be stable for > time Tholdslave)

Sequential Timing

INP -> CL -> Reg

For inp: $T_s = T_{pd,CL} + T_{s,Reg}$

$T_H = T_{H,Reg} - T_{\mathrm{cd,CL}}$

(CL doesn't immediately produce invalid OP once INP turns invalid due to $T_{\mathrm{cd,CL}}$

∴ min hold time for input = $T_{H,Reg} - T_{\mathrm{cd,CL}}$)

For whole circuit: (Tpd & Tcd)

Time taken to produce valid (invalid) OP after CLK rise turns valid (invalid)

R1 -> CL -> OP: $T_{pd} = T_{pd,R1} + T_{pd,CL}$

$T_{cd} = T_{cd,R1} + T_{cd,CL}$

Min CLK Period:

From reg to reg (check all routes for max val)

R1 -> CL -> R2: $T_{CLK,min} = T_{pd,R1} + T_{pd,CL} + T_{s,R2}$

> Ts & Th for inp only concerns paths till 1st reg

> Tpd & Tcd for whole circuit only concerns circuit downstream of last reg b4 output

In Seq Logic, use a **single synchronous clock** (for all flipflops)

1. DD may be violated if users' input not sync w clock edge, invalid values may be stored in "read" mode. This invalid value storing = metastable state.

Metastable State - Properties

> corr to invalid logic level (switching threshold of device), unstable equilibrium (smol △ will cause it to accelerate towards stable 0/1.

> will settle to valid 0/1 eventually (may take aritrarily long)

> EVERY bistable system exhibits at least 1 metastable state

> Inevitable risk of synchronisation (our devices always have a fixed point voltage Vm where Vin=Vm implies Vout=Vm), can only mimimise its probability by introducing more delays between INP & output hopefully signal somehow resolves itself b4 reaching output.

- To fix clock skew (clock does not arrive at all reg at same time, skew is max diff in clk signal arrival times across all FF):

$T_1 = T_{cd,R1} + T_{cd,CL1} > T_{H,R2} + T_{skew}$

$T_2 = T_{pd,R1} + T_{pd,CL} + T_{S,R2} < T_{CLK} + T_{skew}$

FSM has: a set of k states, set of m inp, set of n outputs, transition rules (for each state & inp) & output rules for each state. *A state machines is required when same INP produces multiple diff outputs*

5 states =>ceil $\log_2 5$=3 bits to rep the states. S state bits=2^Spossible states.

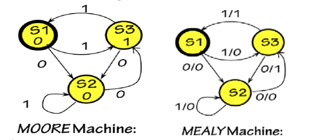
State Transition Diagram:

IN | Current State| Next State| Y

Arcs leaving a state must be mutually exclusive & collectively exhaustive (only 1 choice to go, each inp must have somewhere to go)

FSMs are equivalent iff each INP seq yields same OP seq

Reduce FSM states: 1. 2 states are identical if states have same OP & every INP transit to equivalent states. 2. Find pairs of equivalent states & merge them



MOORE Machine: Outputs on States   *MEALY* Machine: Outputs on Transitions

Implementing FSM as ROM

Given *i* input bits, *s* state bits and *o* output bits, we have 2^{i+s} words and each word has *o* bits as output

Size of ROM = no. of bits needed for the truth table.

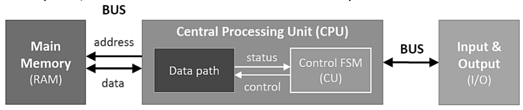
(EG. i|Sn|Sn+1|O => (o+s) 2^{i+s}

No. of different FSM = $2^{(o+s)2^{i+s}}$

FSM Limitations: - finite states (mem) cannot compute parenthesis checker

Von Neumann Model

(INP/OP ↔ CPU ↔ Main Mem)



- **Central Processing Unit**:
1. ALU 2. Control Unit (interprets comp instr) 3. PC (address of current instr)
4. Registers (store temp operands & results)
- **Memory**: storage of N words of W bits. (W is fixed, N vary) Store data & instr. If a word is 32bit, address can be 32 bit and at most 2^{32} entries. 2^{30} is 1GB so at most 4GB entries.
- **INP/OP**: Devices for communicating w outside world
- **Bus** that connects all 3 together

***β* Instructions**: (Low 2 address bits are ignored)

LD (const,rc) = LD (R31,const, rc)

ST (rc,const) = ST(rc,const,R31)

BR(const) = BEQ(R31,const,R31)

FOR BRANCHING: (up to 32767 instr b4/after)

const = (label - <addr of BNE/BEQ>)/4 -1

(no of instr lines b/w br & label instr -1)

(not including BR but including first line of instruction of function 'label')

★ PUSH/POP takes 2 instr

UASM -> interpreted from left to right as least to most significant bytes

. = <address> (lines after this is stored at this address)

y: (labels; symbols that take the value of current mem add)

Interpreter:	Compiler:
- computs exact instr	- Translate high-level lang to low-level assemblr mac lang
- done after execution; slows program execution	- done b4 execution, slows program dev
- decision made during run time, after exec	- Decisions made during compile time

Assembler: takes assembler code outputted by compiler/interpreter and translate into binary machine code.

write 16-bit (WORD) and 32-bit(LONG) words

Little (Big) Endian: least(most) significant bit is stored at lowest memory address

. = 0x0	0x3 0x2 0x1 0x0	0x3 0x2 0x1 0x0
1234	04 03 02 01	01 02 03 04
	(Little used in ***β***)	(Big)

In beta arch, data always comes in 32 bits (not bytes). The address of the entire of each row, by convention, is the smallest byte index of that row, converted into 32 bits.

A 32-bit address is NOT enough to house all 2^{32} permutations of 32-bit of information per row, it can only house $2^{32}/4$ of possible permutatns

Stack & Pointer

- A stack is a concept, a data structure on how we organise data in memory
- **SP**: points to first unused stack location
- **BP**: points to base of the stack (or first item pushed to the stack, registers)
- Add one item a time to top of the stack by push (↑SP by 4 & write data to address as pointed by SP-4)
- Remove one item a time from top of the stack by pop (store the data at address pointed by SP to some reg and reduce the value of SP by 4bytes)

R0 is reserved as the register for the return value of the function getting called

Turing Machine: initial starting state = S0

★ write first then move arrow

- FSM combined with doubly-infinite tape
- Can read & write at tape in every step
- can solve problem with infinitely many states (eg parenthesis checking)

y=Ti[x] ; y is output after arrow executes TM func specs; Ti is TM func spec that dictates machine to go left or right according to state; x is int inp

Church's Thesis: **Every discrete fn computable by ANY realisable machine is computable by some TM.** (Halt fn is incomputable)

Not all well-defined int fn are computable

Universal fn: **U(k,j)=Tk(j)** ; U is computable by a TM; k is input (program to tell us which Tk we want to compute); j is data tape Tk to perform at.

With Tu, don't need to make so many TM to perform each fn cause it can emulate behaviour of any TM (like our comp)

Calling Sequence

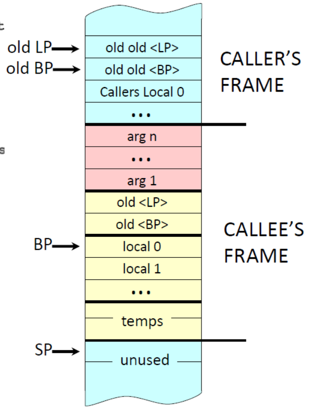
```
        PUSH(arg_n)                  | push args, last arg first
        ...
        PUSH(arg_1)
        BEQ(R31,f, LP)               | Call f.
        DEALLOCATE(n)                | Clean up!
        ...                          | (f's return value in r0)
```

Entry Sequence

```
f:      PUSH(LP)                     | Save LP and BP
        PUSH(BP)                     | in case we make new calls
        MOVE(SP,BP)                  | set BP=frame base
        ALLOCATE(nlocals)            | allocate locals
        (push other regs)            | preserve any regs used
```

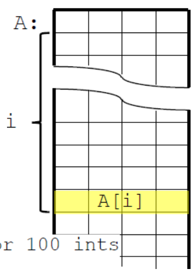
Return Sequence

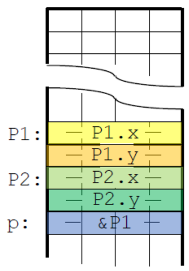
```
        (pop other regs)             | restore regs
        MOVE(val, R0)                | set return value
        MOVE(BP,SP)                  | strip locals, etc
        POP(BP)                      | restore CALLER's linkage
        POP(LP)                      | (the return address)
        JMP(LP,R31)                  | return.
```



Steps for Stack Procedure:

1. PUSH arg in reverse order (1st arg will be at a BP-12)
(write a constant into reg using CMOVE(c,R1) and push(R1))
2. BR(callee,LP): leave return address(last known location+4bytes) in LP
then DEALLOCATE(*args) then HALT()
3. Entry: PUSH(LP), PUSH(BP), MOVE(SP,BP)
4. Exit: MOVE(BP,SP), POP(BP), POP(LP), JMP(LP)
The deallocate is up there at calling seq after BEQ.





C code: β Assembly:

```
int A[100];    A:   .=.+4*100      | Leave room for 100 ints
...                 ...
A[i] += 1;          LD(i,r1)
                    MULC(r1,4,r2)  | index -> byte offset
                    LD(r2,A,r0)    | A[i] -> R0
                    ADDC(r0,1,r0)  | increment
                    ST(r0,A,r2)    | A[i] <- R0
```

2 int input for point structures so they need 8 bytes (2 words) each
y has an offset of 4 bytes because its one row below p.x.

C code: β Assembly:

```
struct Point   P1:  .=.+8
{ int x, y; }  P2:  .=.+8
P1, P2, *p;    p:   .=.+4
               x=0                 | Offset for x component
               y=4                 | Offset for y component
...            ...
P1.x = 157;    ADDC(r31,157,r0)    | r0 <- 157
               ST(r0,P1+x)         | P1.x = 157
...            ...
p = &P1;       ADDC(r31,P1,r3)     | Put &P1 in r3
               ST(r3,p)            | Store r3 in p
p->y = 157;    ST(r0,y,r3)         | p->y = 157;
```

Linkage Pointer: return address to caller (always same for the same recursive call) {usually at deallocate()}

★ Using our procedure linkage convention, how far (in terms of addresses) can a callee be from its caller? > The entry point of the callee must be reachable by a branch, i.e., it must be within approximately 2^15 words of the call instruction. Can overcome by using LDR to load target address into a reg and the JMP using the register.
★ If there is a bound on the no. of states, you can discover its behaviour. For a k-state FSM, every state can be reached in < k steps.
★ 16MB of memory with byte-wise addressing = 16*2^20=2^24addresses
★ If storage for variable is located at address more than 0x8000, (add needs to fit in 15bit) 16-bit constant field isn't large enough so can use LDR to load a 32-bit address into a reg and use LD to fetch data. EG. var: LONG(0x12468)... LDR(var,R0) LD(R0,0,R0)
★ Checking if circuit can be made: Arbitor -> who comes first
if unbounded time: Yes, can make Tpd->bounded
Else: if arbiter: NO
else: YES
★ Branches (b) use a PC-relative displacement while jumps (j) use absolute addresses.
★ Shift left i bits ⇒ $\times 2^i$ (shift right ⇒ $\div 2^i$)
★ BR address that made the call f() is the value of LP-4.

Dec	Hex	Bin
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000

Macro	Definition
BEQ(Ra, label)	BEQ(Ra, label, R31)
BF(Ra, label)	BF(Ra, label, R31)
BNE(Ra, label)	BNE(Ra, label, R31)
BT(Ra, label)	BT(Ra, label, R31)
BR(label, Rc)	BEQ(R31, label, Rc)
BR(label)	BR(label, R31)
JMP(Ra)	JMP(Ra, R31)
LD(label, Rc)	LD(R31, label, Rc)
ST(Rc, label)	ST(Rc, label, R31)
MOVE(Ra, Rc)	ADD(Ra, R31, Rc)
CMOVE(c, Rc)	ADDC(R31, c, Rc)
PUSH(Ra)	ADDC(SP, 4, SP) ST(Ra, -4, SP)
POP(Rc)	LD(SP, -4, Rc) SUBC(SP, 4, SP)
ALLOCATE(k)	ADDC(SP, 4*k, SP)
DEALLOCATE(k)	SUBC(SP, 4*k, SP)