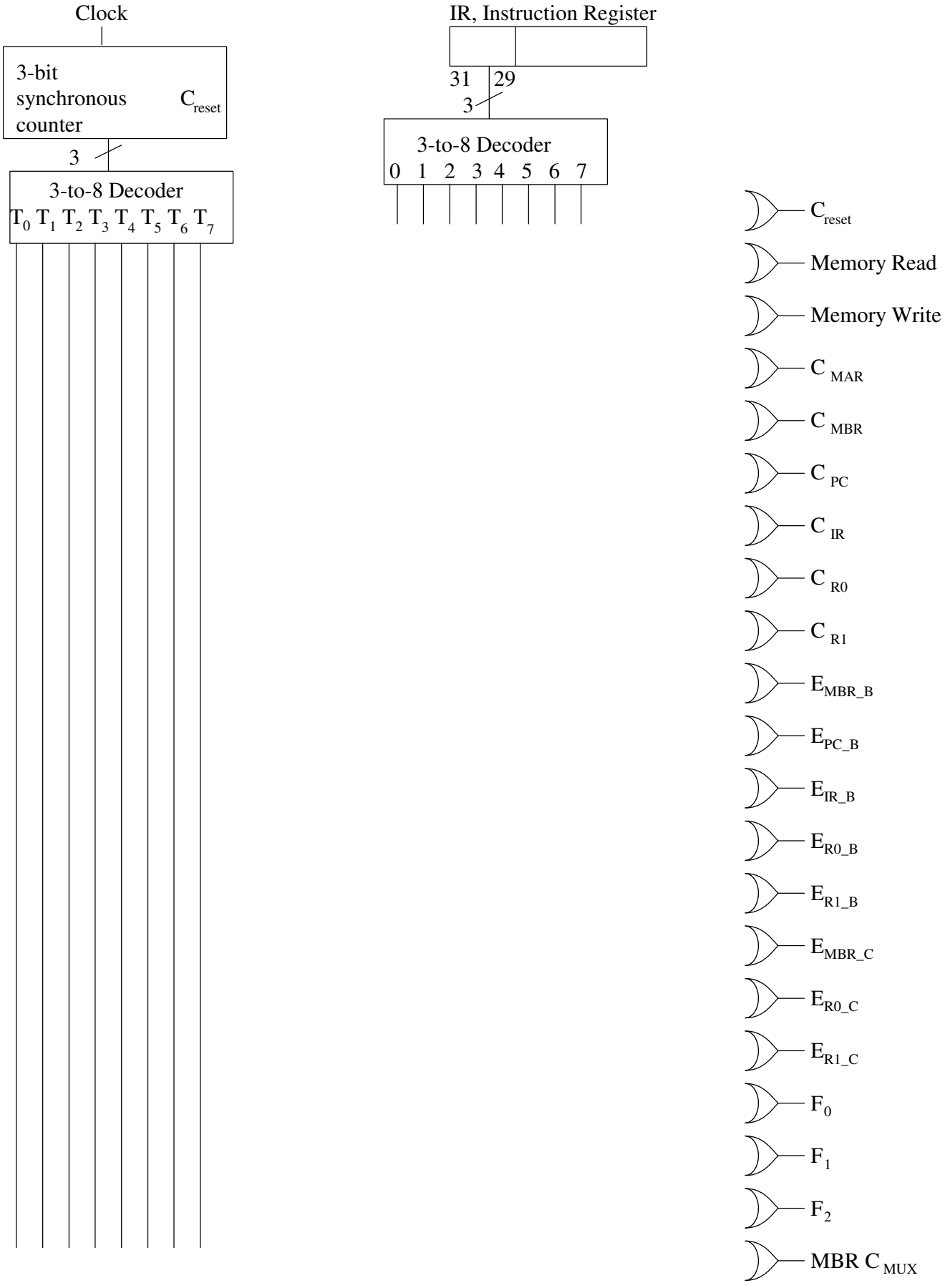
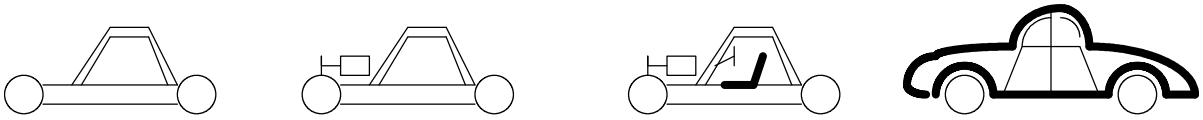
0. Draw the partial combinational logic of the hardwired control unit to handle the LOAD R0, M and STORE M, R1 instructions from lecture 20:



1. Assume that an automobile assembly process takes 4 hours.

Chassis Motor Interior Exterior



a) If the stages take the following amounts of time, then what is the time between completions of automobiles?
Chassis 1 hour Motor 1 hour Interior 1 hour Exterior 1 hour

b) If the stages take the following amounts of time, then what is the time between completions of automobiles?
Chassis 45 minutes Motor 1 hour Interior 1 hour & 15 minutes Exterior 1 hour

2. We could divide the instruction/fetch-execute cycle into stages for instruction pipelined.

- Fetch Instruction - read instruction pointed at by the program counter (PC) from memory into Instr. Reg (IR)
- Decode Instruction - figure out what kind of instruction was read
- Fetch Operands - get operand values from the memory or registers
- Execute Instruction - do some operation with the operands to get some result
- Write Result - put the result into a register or in a memory location

Two approaches for designing a computer is CISC (Complex Instr. Set Computer - pre-1980) and RISC (Reduced Instruction Set Computer post 1985). A CISC philosophy was to make assembly language (AL) as much like a high-level language (HLL) as possible to reduce the "*semantic gap*" between AL and HLL. The rational for CISC at the time was to:

- reduce compiler complexity and aid assembly language programming. Compilers were not too good during the 50's to 70's, (e.g., they made poor use of general purpose registers so code was inefficient) so some programs were written in assembly language.
- reduce the program size. More powerful/complex instructions reduced the number of instructions necessary in a program. Memory during the 50's to 70's was limited and expensive.
- improve code efficiency by allowing complex sequence of instructions to be implemented in microcode. For example, the Digital Equipment Corporation (DEC) VAX computer had an assembly-language instruction "MATCHC *substrLength, substr, strLength, str*" that looks for a substring within a string.

The architectural characteristics of CISC machines include:

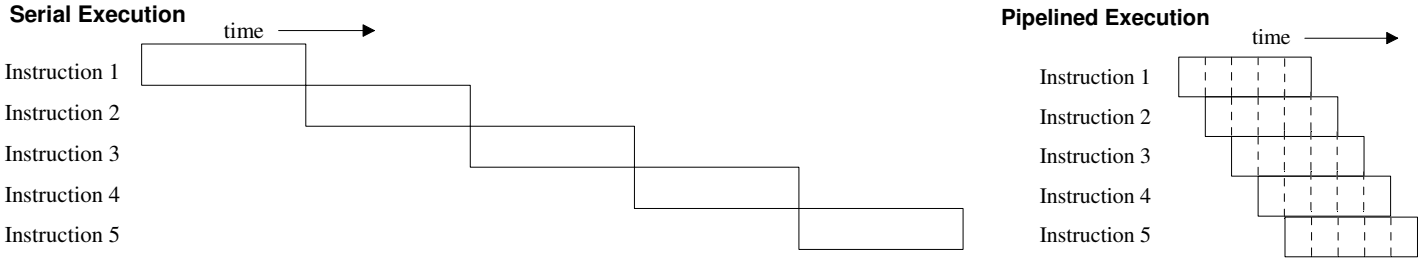
- complex, high-level like AL instructions
- variable format machine-language instructions that execute using a variable number of clock cycles
- many addressing modes (e.g., the DEC VAX had 22 addressing modes)

a) Why are complex instructions of CISC (Complex Instr. Set Computer) machines difficult to pipeline?

b) Why are RISC machines usually Load & Store machines (i.e., only Load and Store instructions access memory)?

3. The whole question refers to a pipelined, RISC machine with five stages:

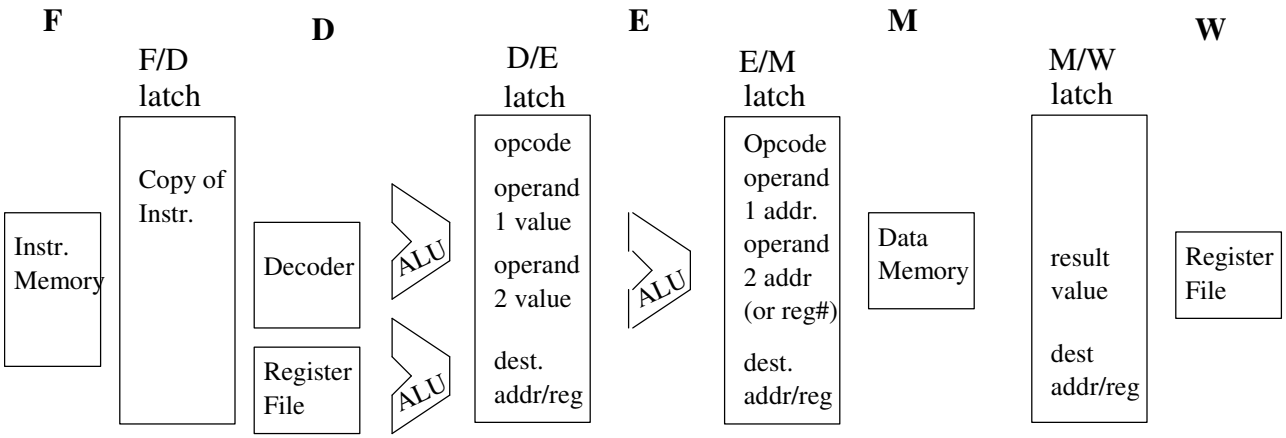
- F, fetch - fetch the instruction from memory
- D, decode - determine the type of instruction **and** read any necessary register values
- E, execute - perform ALU operation or memory address calculation for LOAD or STORE instructions
- M, memory - access memory on LOAD or STORE instruction
- W, write - write register values



Problems that delay/*stall* the pipeline:

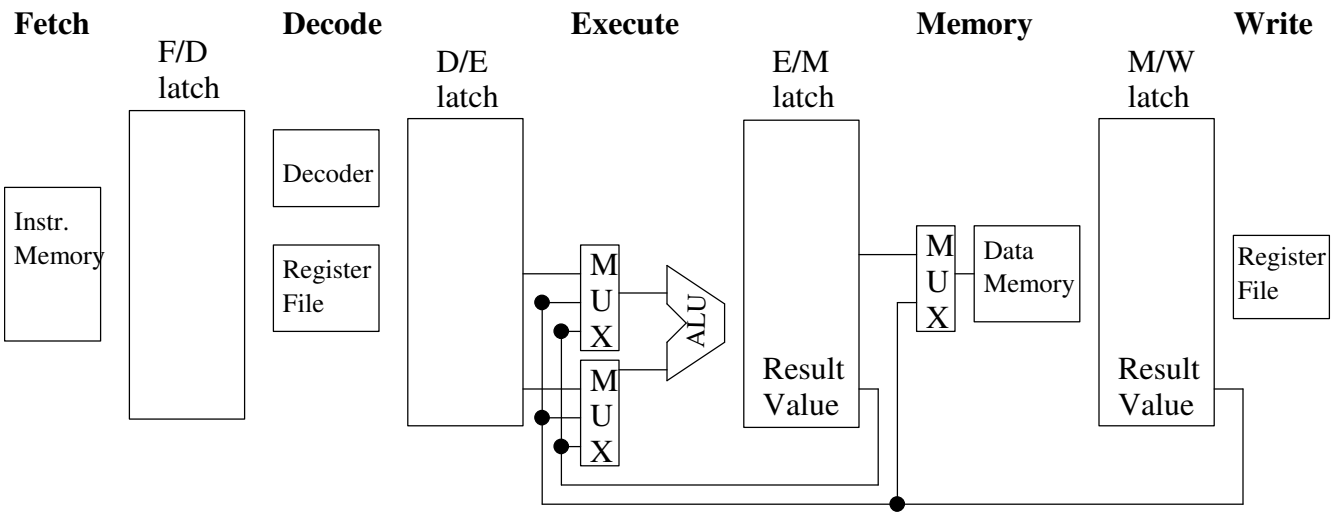
- ***structural hazard -*** a piece of hardware is needed by several stages at the same time, e.g., Memory in F, and M. This might require stages to sequentially access the hardware, or duplicate into two memories.
- ***data hazard*** - an instruction depends on the results of a previous instruction which has not been calculated yet. (RAW) read-after-write example: ADD R3, R2, R1 ; R3← R2 + R1
  SUB R4, R3, R5 ; R4 ← R3 - R5
- ***control/branch hazard*** - branching makes it difficult to fetch the "correct" instructions to be executed

Pipeline latches/registers between each stage. Hold temporary results and act like an IR. Some of the hardware components used (e.g., Memory and Register File) are shown as if they are duplicated, but they are not.



a) Complete the following timing diagram. Insert stalls where necessary (assuming NO by-pass signal paths).

Without by-pass signal paths	**Time →**																			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
ADD R1, R3, R4	F	D	E	M	W															
ADD R2, R4, R5																				
ADD R3, R2, R1																				
LOAD R2, [R3, #8]																				
STORE R2, [R6]																				



b) Complete the following timing diagram assuming by-pass signal paths as shown above.

With by-pass signal paths	**Time →**																			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
ADD R1, R3, R4	F	D	E	M	W															
ADD R2, R4, R5																				
ADD R3, R2, R1																				
LOAD R2, [R3, #8]																				
STORE R2, [R6]																				

4. *Control Hazards* - branching causes problems since the pipeline can be filled with the wrong instructions.

```
IF      BEQ  R3, R8, ELSE
        ADD  R4, R5, R6          /* ADD should not be executed if the branch is taken */
        SUB  R8, R5, R6
        .
        .
        B  END_IF
ELSE    MUL  R3, R3, R2          /* MUL should not be executed if the previous B executes*/
        .
        .
END_IF
```

a) During which stage is the target address (addr. of "ELSE" label) calculated for the BEQ instruction?

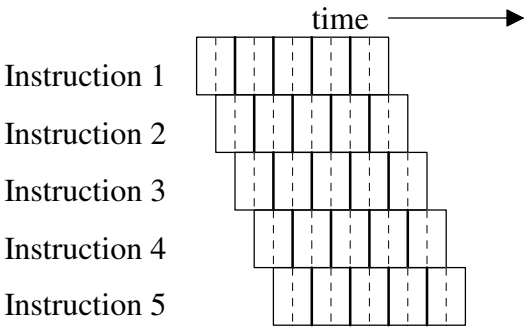
b) During which stage of BEQ instruction is the comparison between registers (R3 and R8) performed (i.e., when is the *outcome* (*taken* or *not taken*) of the branch known)?

If we always (statically) continue to fetch sequentially until the outcome of a conditional branch is known**:**
c) How many cycle branch penalty for a taken outcome?

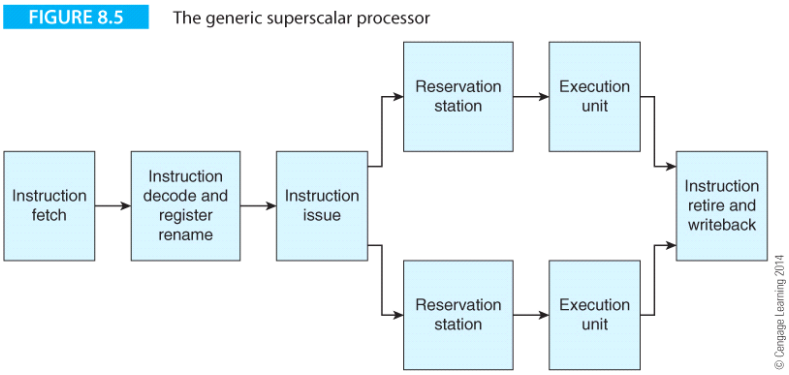
d) How many cycle branch penalty for a not-taken outcome?

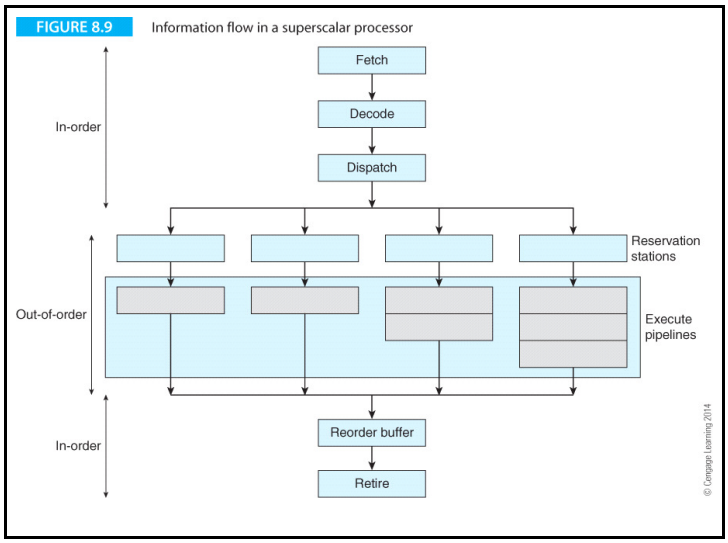
Chapter 8: "Beyond RISC" - goal of multiple instructions completed per clock cycle

superpipelined (e.g., MIPS R4000)- split each stage into substages to create finer-grain stages



superscalar (e.g., Intel Pentium, AMD Athlon)- multiple instructions in the same stage of execution in duplicate pipeline hardware





- Instruction Fetch - obtain "next" instruction(s) from memory (I cache)
- Instruction Decode - decode instr(s) and rename user-visible registers to avoid data hazards (WAW & WAR) introduced by out-of-order execution
  SUB R3, R2, R5
  ADD R4, R3, #1
  ADD R3, R5, #1
  MUL R7, R3, R4

- Instruction *issue* - sent instruction to reservations unit associated with an appropriate execution unit (integer ALU, fl. pt. ALU, LOAD/STORE memory unit, etc.) to await execution
- Reservation station - *dispatch* instruction to execution unit when unit becomes free and all of the instruction's operand values are known
- Instruction retire - writes results of potentially out-of-order instructions back to registers to ensure correct in-order completion. Also, communicates with the reservation stages when instruction completion frees resources (e.g., "virtual" registers used in register renaming)