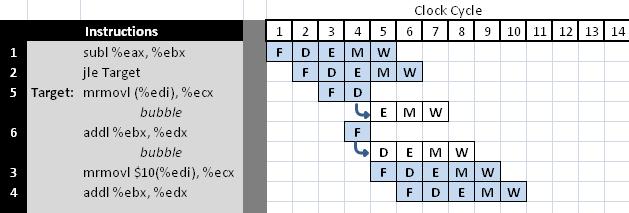
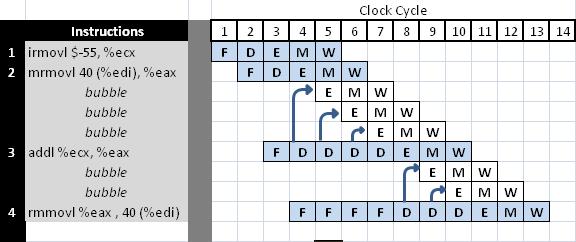
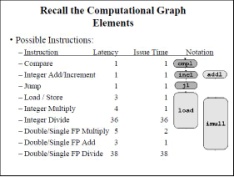
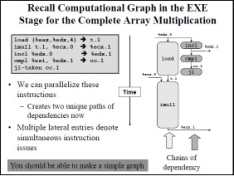
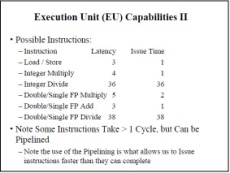
**F:**The program counter returns the current instruction from memory **Extracts the 4-bit instruction code (icode) and the 4-bit function code (ifun)** Extracts the register operands fields ***rA*** and ***rB*** Computes the PC for the next instruction and stores it in ***valP* D:** Reads the appropriate registers from the register file **E:** Computes the register-register ALU operations Computes memory references (adds offset to base addresses) Increments or decrements the stack pointer Tests branch conditions Sets any condition codes **Compute the branch target addresses** If a load, then perform a read using the effective address generated in the EX stage If a store then writes the data using the effective address **W:** Write the results into the register file Do this whether it comes from the memory system (the result of a load) or from the ALU **U:** Updates the program counter to the next instruction to be executed **The hardware actually computes the value in the first stage but does not update until last stage in case of a branch** **Throughput (in GOPS) = 1000 \* 1/Time (in psecs)**

The first thing to do is just lay all the code out in the normal sequential path without worrying about parallelizing the code.  The second thing to do is to check all the inputs and outputs to be sure you did not forget anything.  Each instruction should have one output and they should have one or two inputs depending on the instruction. Once you have done this you are actually in really good shape.  The third step is to then look to see what dependency chains exist in the flow of instructions.  For example, the data dependencies of outputs from one going into the input of another are easy to see now so those instructions should be stacked up.  Then look for other looser dependencies, like loop counter updating where the updating can be done at the same time as something else.  In this you then grab those collections of instructions and see how far up in time you can move them and have them still make sense.

There are two aspects of locality **1. Temporal locality** – if we use a data item, we will probably reuse it in the near future **2. Spatial locality** – if we use a data item we will probably use a data item near to it in memory next. **The Principle of Locality is what makes memory hierarchies work** – Caches hold data items that are close together spatially – The hierarchy of memory keeps data items that have been used closely in time in the fastest memory **There are three approaches to block placement: *1. Direct Mapped***  The block can only be placed in a specific location in the cache ***2. Set-Associative***  A set is a group of blocks in the cache The incoming block can only be placed in a specific set of blocks It can then be placed anywhere within this set ***3. Fully Associative***  A block can be placed anywhere in the cache