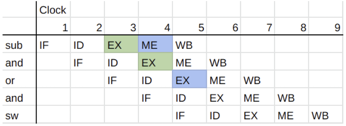
一些文字和图片的手机截图

描述已自动生成图示, 示意图

描述已自动生成一些文字和图片的手机截图

描述已自动生成文本

低可信度描述已自动生成表格

描述已自动生成图示, 示意图

描述已自动生成图示, 示意图

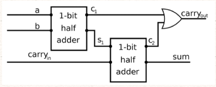
描述已自动生成图示, 示意图

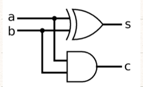
描述已自动生成图示

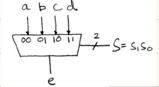
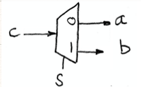
描述已自动生成图示

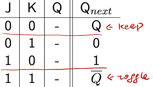
描述已自动生成图示

描述已自动生成**日历

描述已自动生成** **图示

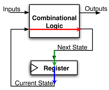
描述已自动生成****CPU performance:** CPU performance determined: latency/throughoutput/responding time •ThroughPut: the total amount of work done in a given unit of time •Clock Frequency/Rate: cycles per second, e.g., 3.0GHz = 3.0\*10^9Hz, ClockRate = 1/ClockCycle •Clock Cycles per Instruction: CPU time = #Instruction\*CPI\*CC or /CR **Locality:** •Temporal: recently accessed item are likely to be accessed again in the near future(loops, repeated call to same function) •Spatial: Recently accessed items are likely to be in continue ram space(array, sequential operations) •Continually using the same type of instruction is not locality  **Caching: •Hit:** find b in the cache at level k •**Miss:** not find at level k, so the level k cache must fetch b from lower level k+1 •If level k is full then some block must be replaced •Mapping: apply modding (i.e., 7mod4=3) to get position. •Replacement: if it is full in mapping step, use either LeastRecentlyUsed or FirstInFirstOut to overwrite contents. • Cold miss: when the block b does not exist at level k. It occurs while data is caching for the first time. • Capacity miss: when active block is larger than the size of cache • Conflict miss: when multiple data from level k+1 map to same position in level k due to mapping policy, it may cause trashing Example: 2-way set, 4 bytes cache lines, 4 sets, LRU policy, cache: 4,6,26,12,0,2,21,31. •4=0100 =>index=10, offset=00(cold miss) •6=0110 =>index=10, offset=10(hit), •26=11010 =>index=10, offset=10(cold miss) •31=11111 =>index=10, offset=11 (hit) **AMAT Calculation:**  Example: Main mem access=100ns, 50%instruction require data access, L1miss rate=3.6%, hit time=1.26ns, idealCPI=2.0, L2 miss rate=42%, hit time=21.24ns •**AMAT1** =HitTime+(MissRate\*Penalty) =1.26+(3.6%\*100) =4.86ns •**CPIstall1** =CPIideal+AccessRate\*MissRate\*(MissPenalty/HitTime) =2+3.6%\*50%\*(100/1.26) =3.43 •**CPUTime2** =IC\*CPIstall\*ClockCycle =IC\*3.43\*1.26 =IC\*4.32 •**AMAT2** =L1HitTime+L1MissRate\*(L2HitTime+L2MissRate\*Penalty) =1.26+3.6%\*(21.24+42%\*100) =3.54ns •**AvgMemStallCycle2** =AccessRate\*L1MissRate\*(L2HitTime+L2MissRate\*L2MissPenalty) =50%\*3.6%\*(21.24+42%\*100/1.26) =0.98 **•CPIstall2** =CPIideal+AMSC =2+0.98 =2.98 •**CPUTime2** =IC\*CPIstall\*CC =IC\*2.98\*1.26=IC\*3.75 •**CNF**: (a+b)\*(c+d)\*(e+f) •**DNF**: ab+cd+ef •Functional Completeness: a sets of functions which can describe every operations. **{+, \*, not} is functional completeness. {+, not} is bothe complete and minimum.**

**•1 bit half adder:** S = A XOR B C = A AND B **•1bit full adder:** S = (A XOR B) XOR Cin **n-bit full adder:** C0in = C1out, connection of 1 bit full adders **MUX & DEMUX:** n-bit ctrl signal => 2^n options

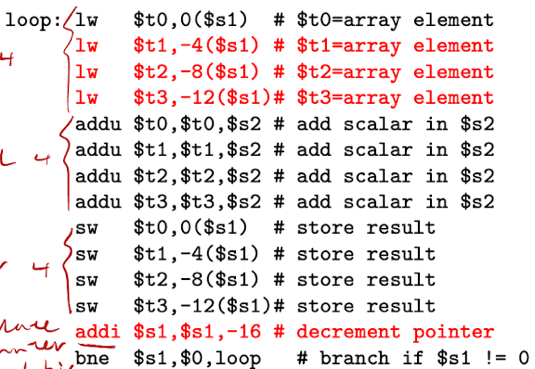
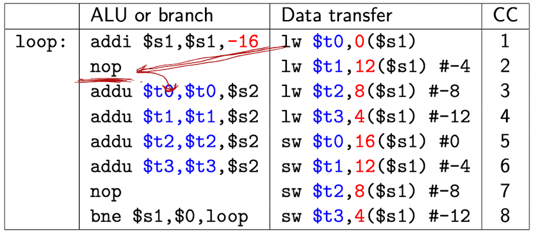
**手机屏幕截图

描述已自动生成手机屏幕截图

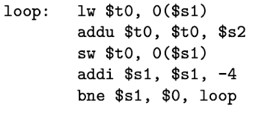
中度可信度描述已自动生成**

手机屏幕的截图

描述已自动生成表格

描述已自动生成文本

描述已自动生成•**D**: do nothing, just delay for a cycle **•T:** if input = 1, toggle current state •**SR:** S=1 set next state 1, R=1 set next state 0; S=0 and R=0 hold; Cannot have S=1 and R=1 **•JK:** Same as SR, but toggle if j=k=1. **•Registers** are just collection of flip-flops, n-bits => n- registers: ParallalInParallalOut, SerialInParallalOut, SISO, PISO Min. Clock Cycle = **Combinational circuit propagation delay** + **setup time** + **clk-to-q** •Speedup: Split add and shift into two different tasks / Insert register between to store results temporarily / increase clock frequency  **MIPS Assembly: RTL •** add $8, $9, $10 = R[8] <- R[9] + R [10] • addi $8, $9, 127 = R[8] <- R[9] +127 • lw $13, 32($10) = R[13] <- Mem[R[10] + 32] • sw $13, 8($10) = Mem[R[10] + 8] <- R[13] **addui**: i stands for immediate values, u stand for unsign(force handling data unsign. addui $t0, $t1, -1 is actually adding the maximum unsigned value due to underflow) \*: words and int are 32-bits **Instruction Formats: •R-type:** opcode(6: 000000), rs(5, first param), rt(5, second param), rd(5, destination), shamt(5, shift amount), funct (6, funct code) **•I-type:** opcode(6), rs(5, source), rd(5, destination), imm(16, offset/value for arithmetic operations) •**J-type:** opcode (5, j=000010, jal=000011), address(26, pseudo-direct: [next\_PC=(PC&0xF0000000)|(target << 2))], PC-relative jump are bne/beq in I-type) **R** add $t0, $s1, $s2: 000000|1001|10010|01000|00000|100000 **R** sll $s0, $t0, 4: 000000|00000|01000|10000|00100|000000 **I** addi $t1, $t0, 10: 001000|01000|01001|0000000000001010 **I** lw $t1, 12($t0): 100011|01000|01001|0000000000001100 **I** bne $t0, $t1, 24: 000101|01000|01001|0000000000011000 <= this is pc-relative

 **VILW**: **very long instruction words,** encoded multiple instructions into an issue package. E.g., 64bit package=(32bit)+(32bit), one IF stage for one package, multiple ID, EX simultaneous **Renaming:**  for every instruction which **write to a register,** create a new value name for the destination. add t6, t0, t2 sub t4, t2, t0 xor t0, t6, t2 and t2, t2, t6  **MultiThreading:** software, programming model which allows for multiple, concurrent threads execution, each with own context(variables, PC, register value, stack), **but share address space heap**. Multithread does not require multicore. If no multicore, multithreading could be achieved via time-division multiplexing, requiring context switching. With multicore, threading could run simultaneously. multicore: hardware, multiprocessor: hardware **Superscalar:** multiple instructions within a clock cycle. Multiple ALU in data path running same time.

**图片包含 图示

描述已自动生成**表格

描述已自动生成•**Multi cycle data path minimum clock ------------------------------------------------------------ --------------------**Single cycle datapath: 800ps clock cycle, pipelined: 200ps clock cycle, uneven time for each stage: ID and WB only 100 ps. Assuming 3 lw instruction above, Ideal speedup = time between instructions in single cycle/in pipelined = 800/200 = 4 Actual speedup=time complete in single cycle/in pipelined=(3\*800)/(7\*200)=1.714 Actual speedup is approaching ideal speedup as the number of instructions increase **Classic Performance time = IC\*CPI\*Clock cycle**  **Time for pipeline = fill time + (IC\*clock cycle) => CPI approaching 1 Structural Hazard:** two instructions need to use same hardware at same time, **Data Hazards:** RAW: the only depenciy; RAR: never hazard; WAR: most time not hazard; WAW: out-of-order execution. **Forwarding:** ALU-ALU, ALU-MEM, MEM-MEM **VLIW:** •Static Scheduling •In-order execution •single IF but many EX units •Instruction packed together in issue packet by compiler **Static SS:** •Static scheduling •in-order execution •many IF units(or one IF fetching multiple instr.) and many EX units •Compiler explicitly schedules each “route” through the datapath. **Dynamic SS:** dynamic scheduling •out-of-order exec. •singe IF unit but many EX units •IF unit might fetch multiple instr. Per cycle