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

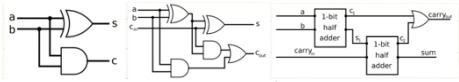
Locality:
•Temporal: recently accessed item are likely to be accessed again in the near future(loops, repeated call to same function) •Spatial: Recently accessed items are likely to be in continue ram space(array, sequential operations) •Continually using the same type of instruction is not locality

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

00	0	1	**2**	3
01	4	5	**6**	7
	20	21	22	23
10	24	25	26	27
11	12	13	14	15
	28	29	30	31

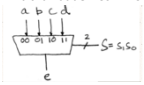
Example: 2-way set, 4 bytes cache lines, 4 sets, LRU policy, cache: 4,6,26,12,0,2,21,31. • 4=0100 =>index=10, offset=00(cold miss) • 6=0110 =>index=10, offset=10(hit), • 26=11010 =>index=10, offset=10(cold miss) •31=11111 =>index=10, offset=11 (hit)

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

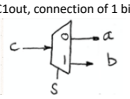


•CNF: (a+b)*(c+d)*(e+f) **•DNF**: ab+cd+ef •Functional Completeness: a sets of functions which can describe every operations. **{+, *, not} is functional completeness. {+, not} is bothe complete and minimum.**

•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



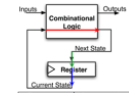
D	Q	Q_{next}
0	-	0
1	-	1

T	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	0

S	R	Q	Q_{next}
0	0	-	Q
0	1	-	0
1	0	-	1
1	1	-	X

J	K	Q	Q_{next}
0	0	-	Q
0	1	-	0
1	0	-	1
1	1	-	$\bar{Q}$

•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

Category	Instruction		OP/ funct	Example	Meaning
Logic & Arith.	add	R	0/32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	R	0/34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	I	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	and/or	R	0/(36/37)	(and/or) $s1, $s2, $s3	$s1 = $s2 (∧/∨) $s3
	(and/or) immediate	I	12/13	(andi/ori) $s1, $s2, 6	$s1 = $s2 (∧/∨) 6
	shift right logical	R	0/2	srl $rd, $rt, 4	$rd = $rt >> 4
	shift right arithmetic	R	0/3	sra $rd, $rt, 4	$rd = $rt >> 4
Data Transfer	load word	I	35	lw $s1, 24($s2)	$s1 = Memory($s2+24)
	store word	I	43	sw $s1, 24($s2)	Memory($s2+24) = $s1
	load byte	I	32	lb $s1, 25($s2)	$s1 = Memory($s2+25)
	store byte	I	40	sb $s1, 25($s2)	Memory($s2+25) = $s1
Cond. Branch	br on equal	I	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	I	5	bne $s1, $s2, L	if ($s1 != $s2) go to L
	set less than	R	0/42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set less than immediate	I	10	slti $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
Uncond. Jump	jump	J	2	j 250	go to 1000
	jump register	R	0/8	jr $t1	go to $t1
	jump and link	J	3	jal 250	go to 1000; $ra=PC+4

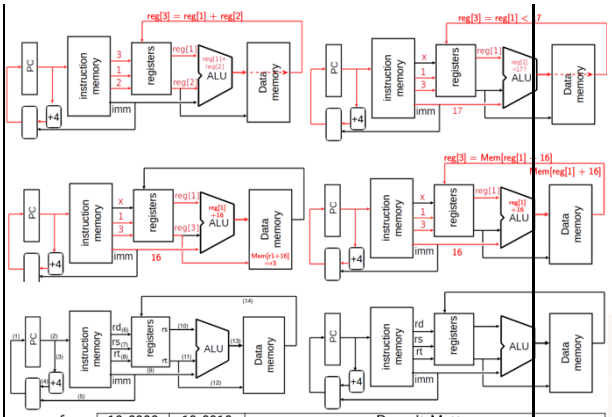
MIPS Assembly: RTL
• add $8, $9, $10 = R[8] <- R[9] + R [10] • addi $8, $9, 127 = R[8] <- R[9] +127 • lw $13, 32($10) = R[13] <- Mem[R[10] + 32] • sw $13, 8($10) = Mem[R[10] + 8] <- R[13]
addui: i stands for immediate values, u stand for unsign(force handling data unsign.

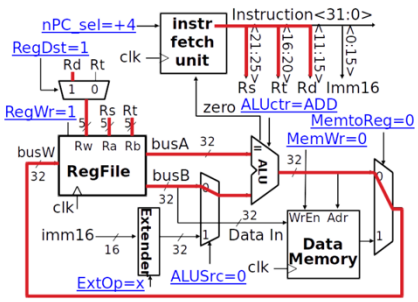
```
    void swap(int v[], int k) {
        int temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
    }
```

addui $t0, $t1, -1 is actually adding the maximum unsigned value due to underflow)
*: words and int are 32-bits

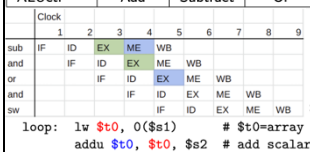
```
swap:  sll $t1, $a1, 2     # $t1 = k * 4
       add $t1, $a0, $t1   # $t1 = v+(k*4)
                           # (address of v[k])
       lw $t0, 0($t1)      # $t0 (temp) = v[k]
       lw $t2, 4($t1)      # $t2 = v[k+1]
       sw $t2, 0($t1)      # v[k] = $t2 (v[k+1])
       sw $t0, 4($t1)      # v[k+1] = $t0 (temp)
       jr $ra              # return to calling routine
```

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





func	10 0000	10 0010	Doesn't Matter				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
nPC_sel	0	0	0	0	0	1	?
Jump	0	0	0	0	0	0	1
ExtOp	x	x	1	1	1	x	x
ALUctr	Add	Subtract	Or	Add	Add	Equal	x




```
sub  $t2, $t1, $t3
and  $t7, $t2, $t5
or   $t8, $t6, $t2
add  $t9, $t2, $t2
sw   $t5, 12($t2)
```

```
loop: lw $t0, 0($s1)     # $t0=array element
      addu $t0, $t0, $s2 # add scalar in $s2
      sw $t0, 0($s1)     # store result
      addi $s1, $s1, -4  # decrement pointer
      bne $s1, $0, loop  # branch if $s1 != 0
```

	ALU or branch	Data transfer	CC
loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1, -4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $0, loop	sw $t0, 4($s1)	4

- Scheduled 5 instructions in 4 cycles. In the limit, CPI approaches 0.8.
- nops don't count towards performance.
- Sometimes when scheduling code you need to adjust offsets.

I lw $t1, 12($t0): 100011|01000|01001|0000000000001100
I bne $t0, $t1, 24: 000101|01000|01001|0000000000011000 <= this is pc-relative

```
loop:  lw $t0, 0($s1)
       addu $t0, $t0, $s2
       sw $t0, 0($s1)
       addi $s1, $s1, -4
loop:  lw   $t0,0($s1)   # $t0=array element
       lw   $t1,-4($s1)  # $t1=array element
       lw   $t2,-8($s1)  # $t2=array element
       lw   $t3,-12($s1)# $t3=array element
       addu $t0,$t0,$s2 # add scalar in $s2
       addu $t1,$t1,$s2 # add scalar in $s2
       addu $t2,$t2,$s2 # add scalar in $s2
       addu $t3,$t3,$s2 # add scalar in $s2
       sw   $t0,0($s1)  # store result
       sw   $t1,-4($s1) # store result
       sw   $t2,-8($s1) # store result
       sw   $t3,-12($s1)# store result
       addi $s1,$s1,-16 # decrement pointer
       bne  $s1,$0,loop # branch if $s1 != 0
```

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.

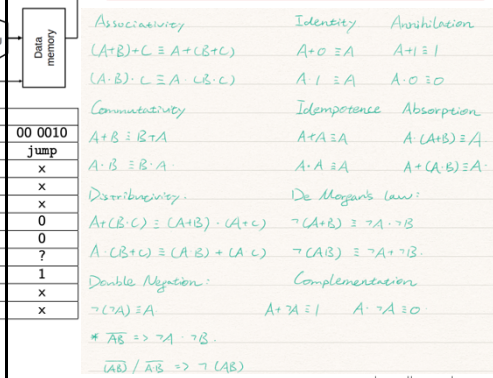
Instr.	$t0	$t2	$t4	$t6	Renamed Instr.
Initially	V0	V1	V2	V3	—
add $t6, $t0, $t2				V4	add V4, V0, V1
sub $t4, $t2, $t0			V5		sub V5, V1, V0
xor $t0, $t6, $t2	V6				xor V6, V4, V1
and $t2, $t2, $t6		V7			add V7, V1, V4

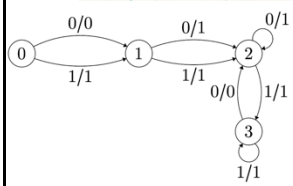
add t6, t0, t2
sub t4, t2, t0
xor t0, t6, t2
and t2, t2, t6

	ALU or branch	Data transfer	CC
loop:	addi $s1,$s1,-16	lw $t0,0($s1)	1
	nop	lw $t1,12($s1) #-4	2
	addu $t0,$t0,$s2	lw $t2,8($s1) #-8	3
	addu $t1,$t1,$s2	lw $t3,4($s1) #-12	4
	addu $t2,$t2,$s2	sw $t0,16($s1) #0	5
	addu $t3,$t3,$s2	sw $t1,12($s1) #-4	6
	nop	sw $t2,8($s1) #-8	7
	bne $s1,$0,loop	sw $t3,4($s1) #-12	8

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.



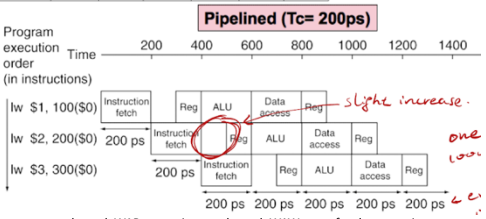


PS	In	NS	Out
00	0	01	0
00	1	01	1
01	-	10	1
10	0	10	1
10	1	11	1
11	0	10	0
11	1	11	1

•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

Instr.	IF	ID	EX	MEM	WB	Total
R-type	200ps	100ps	200ps	-	100ps	600ps
Branch	200ps	100ps	200ps	-	-	500ps
sw	200ps	100ps	200ps	200ps	-	700ps
lw	200ps	100ps	200ps	200ps	100ps	800ps

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