A · LOGIC BLOCKS

4→2 Priority Encoder (D3 highest)

D3 D2 D1 D0	Y1 Y0	V
0 0 0 0	x x	0
0 0 0 1	0 0	1
0 0 1 x	0 1	1
0 1 x x	1 0	1
1 x x x	1 1	1

Y1 = D3 + D2 Y0 = D3 + D2'·D1 V = D3+D2+D1+D0

Magnitude Comparator (4-bit, A=a3..a0, B=b3..b0)

$E_i = a_i b_i + a_i' b_i' = a_i$ XNOR b_i ("bit i equal")

EQ = E3·E2·E1·E0

GT = a3b3' + E3·a2b2' + E3E2·a1b1' + E3E2E1·a0b0'

LT = a3'b3 + E3·a2'b2 + E3E2·a1'b1 + E3E2E1·a0'b0

Iterative stage (MSB→LSB, MSB stage gets in_gt=0, in_eq=1, in_lt=0):
out_gt = in_gt + in_eq·a·b' | out_lt = in_lt + in_eq·a'·b | out_eq = in_eq·(a XNOR b)
8-bit = 2×4-bit chained (hi outputs → lo inputs).
MIN(A,B) = 2:1 mux, sel = AgtB (A>B → pick B).

B · FLOATING POINT

IEEE formats value = $(-1)^{s} \times 1.f \times 2^{e-bias}$

Format	s/e/f	bias	notes
Single (float)	1/8/23	127	max≈3.4e38, min norm 2^{-126}
Double	1/11/52	1023	max≈1.8e308
Lab4 17-bit	1/5/11	15	e=1..30 normal; max= $(2-2^{-11})\cdot 2^{15}$=65520; min norm 2^{-14}; min denorm 2^{-25}

Bias = $2^{k-1}-1$. Ex (17-bit): 1.5 = 0 01111 10000000000 = 0x07C00; 0.75 = 0 01110 10000000000.
Float encodings: 1.0=3F800000, −8=C1000000, 0.1=3DCCCCCD, 12.3=4144CCCD, 2.0=40000000

Special cases

Exponent	Fraction	Meaning
0	0	±0
0	≠0	Denormal: $(-1)^{s}\times 0.f\times 2^{1-bias}$ (no hidden 1)
1..max−1	any	Normal (hidden 1)
all 1s	0	±∞ (overflow result)
all 1s	≠0	NaN (∞−∞, 0/0, NaN op x)

FP Add/Sub algorithm

1. **Unpack**: s, e, m = 1.f (0.f if e=0). Sub: flip s_B, then add.
2. **Swap** so |A| ≥ |B| (compare e, then m). d = $e_A - e_B$.
3. **Align**: shift m_B right by d (keep guard/round/sticky bits). e = e_A.
4. **Add/sub** mantissas: same sign → m_A+m_B; diff sign → m_A-m_B. Result sign = s_A.
5. **Normalize**: carry out (m≥2) → shift right 1, e+1. Leading zeros k → shift left k, e−k.
6. **Round** (nearest-even via G,R,S) → may need renormalize.
7. **Check**: e ≥ all-1s → overflow → ∞; e ≤ 0 → underflow → denorm/0; m=0 → result 0 (e=0).

HW: 5-bit exp subtractor → swap muxes → right barrel shifter → 12-bit add/sub (11+hidden) → leading-zero detector + left shifter → exponent ± adjust.
Ex: 1.5+0.75: $1.10\times2^{0} + 1.10\times2^{-1}$ → 1.10 + 0.11 = 10.01 → 1.001×2^{1} → 0 10000 00100000000 = 2.25

D · MIPS ASSEMBLY

Registers

Name	#	Use	Saved by
$zero	0	always 0	–
$at	1	assembler temp	–
$v0–$v1	2–3	return vals / syscall code	caller
$a0–$a3	4–7	args 1–4	caller
$t0–$t7	8–15	temps	caller
$s0–$s7	16–23	saved	**callee**
$t8–$t9	24–25	temps	caller
$k0–$k1	26–27	OS kernel	–
$gp	28	global ptr 0x10008000	–
$sp	29	stack ptr 0x7fffeffc	**callee**
$fp	30	frame ptr	**callee**
$ra	31	return addr (jal)	**callee**

Caller-saved ($t,$a,$v): callee may trash → caller saves if needed after call. **Callee-saved** ($s,$sp,$ra): if function uses it, it must restore.
FP: $f0-$f31, return $f0, arg $f12.

Instruction formats

```
R:
op(6)|rs(5)|rt(5)|rd(5)|shamt(5)|funct(6)  op=0
I: op(6)|rs(5)|rt(5)|imm(16)
J: op(6)|address(26)
```

R: `add rd,rs,rt` · I: `addi rt,rs,imm` / `lw rt,imm(rs)` / `beq rs,rt,lbl` · shift: `sll rd,rt,shamt` (rs=0)

Opcode / funct (hex)

R-type (op 0) funct		I/J-type opcode	
sll 00 srl 02	sra 03	j 02	jal 03
jr 08 jalr 09	syscall 0c	beq 04	bne 05
mfhi 10 mflo 12	mult 18	blez 06	bgtz 07
multu 19 div 1a	divu 1b	addi 08	addiu 09
add 20 addu 21	**sub 22** subu 23	slti 0a	sltiu 0b
and 24 **or 25**	xor 26 nor 27	andi 0c ori 0d	xori 0e lui 0f
slt 2a	sltu 2b	lb 20 lh 21	**lw 23** lbu 24
mul (op 1c, funct 02)		sb 28 sh 29	**sw 2b** lwc1 31 swc1 39

Branch: target = PC+4 + (SignExt(imm)≪2) → imm = (target − (PC+4))/4 (in words)
Jump: target = {(PC+4)[31:28], addr26, 00} → addr26 = target≫2

Encoding examples

```
addi $2,$0,5  001000 00000 00010
0000000000000101 = 20020005
add $5,$5,$4  000000 00101 00100 00101
00000 100000 = 00a42820
beq $5,$7,end @0x18: op4 rs5 rt7 imm=
(0x44−0x1c)/4=10 = 10a7000a
sw $7,68($3)  101011 00011 00111
0000000001000100 = ac670044
j end (0x44)  000010 addr=0x44>>2=0x11
= 08000011
```

Syscalls ($v0 = code, then `syscall`)

$v0	Service	Args / result
1	print int	$a0
2	print float	$f12
3	print double	$f12
4	print string	$a0 = addr of asciiz
5	read int	→ $v0
6 / 7	read float / dbl	→ $f0
8	read string	$a0 buf, $a1 max len
9	sbrk (alloc)	$a0 bytes → $v0 addr
10	exit	–
11	print char	$a0
12	read char	→ $v0
17	exit2 (with code)	$a0

Data directives & memory

.byte	1 B	.half 2 B (align 2)
.word	4 B (align 4)	.word 0:10 → 10 zero words
.float	4 B	.double 8 B (align 8)
.ascii	no null	.asciiz adds 0x00
.space n	n bytes, no align	.globl main / .align k

.text 0x00400000 · .data 0x10010000 · heap 0x10040000 · $gp 0x10008000 · $sp 0x7fffeffc
Little-endian: `.word 0x12345678` at A → A:78 A+1:56 A+2:34 A+3:12. MARS word view shows byte3..byte0.
ASCII: '0'=30 'A'=41 'a'=61 ' '=20 '\n'=0a NUL=00

Lab7 Code 0 layout

Symbol	Addr	Contents
var1	10010000	41 45 7F FF 0A → word0=FF7F4541
var2	10010006	F6 FF, FF FF (pad 05) → word1=FFF6000A
var3	1001000C	12345678
Var4	10010010	40 B zeros (→0x37)
var5	10010038	4144CCCD, 3DCCCCCD
var6	10010040	3FB999999999999A (0.1 dbl)
str1	10010048	9 B "A String\n"
str2	10010051	23 B (22 chars + NUL)
array	10010068	100 B
name	100100CC	

Code 6: A@10010000=0x20, B@04=0x10, Sum@08=0x30 (48), Diff@0C=0x10 (16), test@10: 3DCCCCCD 3E4CCCCD, testd@18.

Pseudo-instruction expansions

Pseudo	Real
move $d,$s	addu $d,$0,$s
li $t,small	addiu $t,$0,imm
li $t,big	lui $at,hi ; ori $t,$at,lo
la $t,lbl	lui $at,0x1001 ; ori $t,$at,off
lw $t,lbl	lui $at,0x1001 ; lw $t,off($at)
blt a,b,L	slt $at,a,b ; bne $at,$0,L
bgt a,b,L	slt $at,b,a ; bne $at,$0,L
ble a,b,L	slt $at,b,a ; beq $at,$0,L
bge a,b,L	slt $at,a,b ; beq $at,$0,L
not $d,$s	nor $d,$s,$0
neg $d,$s	sub $d,$0,$s
b L / beqz	beq $0,$0,L / beq $s,$0,L

slt vs sltu: $t0=0xFFFFFFFF, $t1=1 → slt=1 (−1<1), sltu=0 (4294967295>1). slti/sltiu = immediate versions. Don't use $at yourself.

Control-flow templates (branch on the OPPOSITE)

```
# if (a==b) x=10;            # if-else
  bne $t0,$t1,done            bne
$t0,$t1,ELSE
  li  $t2,10                  li $t2,10
done:                         j  DONE
# don't forget!
                             ELSE: li
$t2,20
                             DONE:
# if (a<b) via slt           # AND:
if(a>0 && a<100)
  slt $t3,$t0,$t1             ble
$t0,$zero,DONE
  beq $t3,$zero,done          bge
$t0,100,DONE
  li  $t2,100                 li  $t1,1
done:                        DONE:
# OR: if(x<0 || x>100)       # even/odd
  blt $t0,$zero,INV           andi
$t1,$t0,1
  bgt $t0,100,INV             bne
$t1,$zero,ODD
  j DONE
INV: li $t1,1
DONE:
# for(i=0;i<n;i++)           #
while(x!=0)
  li $t0,0                   W: beq
$t0,$zero,END
L: bge $t0,$a1,END             ...
   ...                         j W
   addi $t0,$t0,1            END:
   j L
END:
```

Functions: jal / jr $ra / jalr

`jal f`: $ra=PC+4, jump. `jr $ra`: return. `jalr $t0`: call addr in reg (fn pointer). `j` does NOT save $ra. Args $a0-$a3 → result $v0. **Leaf** (no calls) needn't save $ra; **non-leaf** must (every jal overwrites $ra). Stack grows DOWN; 1 word = 4 B.

```
func:                          # prologue
  addi $sp,$sp,-12
  sw   $ra,8($sp)
  sw   $s0,4($sp)
  sw   $s1,0($sp)
  ...  body (jal other) ...
  lw   $s1,0($sp)              # epilogue
  lw   $s0,4($sp)
  lw   $ra,8($sp)
  addi $sp,$sp,12
  jr   $ra
# push: addi $sp,$sp,-4 ; sw
$t0,0($sp)
# pop : lw $t0,0($sp)  ; addi
$sp,$sp,4
```

Recursive factorial

```
factorial:                     # n in
$a0, n! in $v0
  ble  $a0,1,base
  addi $sp,$sp,-8
  sw   $ra,4($sp)
  sw   $a0,0($sp)
  addi $a0,$a0,-1
  jal  factorial               # $v0=(n-
1)!
  lw   $a0,0($sp)
  lw   $ra,4($sp)
  addi $sp,$sp,8
  mul  $v0,$a0,$v0             # n*(n-1)!
  jr   $ra
base: li $v0,1
  jr   $ra
# iterative
fact: li $v0,1
L:  ble  $a0,1,D
    mul  $v0,$v0,$a0
    addi $a0,$a0,-1
    j L
D:  jr $ra
```

12! = 479001600 is max fitting int32; 13! overflows (mul wraps silently).

Fibonacci: recursive vs iterative (Lab 9)

```
fib:  # iterative, N in $a0
  or   $v0,$zero,$zero ; beq
$a0,$zero,end
  ori  $v0,$zero,1     ; beq
$a0,$v0,end
  or   $t2,$zero,$zero    # prev2=0
  ori  $t1,$zero,1        # prev1=1
  addi $a0,$a0,-1
loop: add $v0,$t1,$t2     #
fib=prev1+prev2
  or   $t2,$zero,$t1
  or   $t1,$zero,$v0
  addi $a0,$a0,-1
  bgtz $a0,loop
end: jr $ra
```

- **Recursive**: calls = 2·F(N+1)−1 → instr count ~ ϕ^N (φ≈1.618), exponential. Each non-base call: 8 mem ops (sw/lw $ra,$s0,$s1,$s2) + 2 jal + 1 jr. Saves $ra on stack.
- **Iterative**: ~5 instr/iter → linear O(N), **0 memory ops** (leaf, registers only).
- **Errors/gotchas**: entry label is `__start` not `main`; `or $v0,$zero,1` should be `ori`; nops are delay-slot fillers (MARS delayed branching off → wasted instrs); F(47)=2971215073 > $2^{31}-1$ → add traps overflow when N≥47 (iterative runs to 49); recursive N=20+ is very slow.
- Measure: MARS → Tools → Instruction Statistics → *Connect to MIPS* → run. Categories: ALU, jump, branch, memory, other.

Arrays: b[i] = a[a[i]] (i=0..99)

```
  la   $s0,a ; la $s1,b ; li $t0,0
L: slti $t1,$t0,100
  beq  $t1,$zero,DONE
  sll  $t2,$t0,2           # i*4
  add  $t3,$s0,$t2
  lw   $t4,0($t3)          # a[i]
  sll  $t4,$t4,2           # a[i]*4
  add  $t4,$s0,$t4
  lw   $t5,0($t4)          # a[a[i]]
  add  $t6,$s1,$t2
  sw   $t5,0($t6)          # b[i]=
  addi $t0,$t0,1
  j L
DONE:
```

Pass array to function: $a0 = base addr (la), $a1 = count; walk with `addi $a0,$a0,4`.

Cross-compilation

```
sudo apt-get install gcc-mips-linux-
gnu cpp-mips-linux-gnu
mips-linux-gnu-gcc -O3 -S -mfp32 -
march=R2000 code.c  → code.s
mips-linux-gnu-g++ -O3 -S -mfp32 -
march=R2000 code.cpp
clang -O3 -S -target mips -mcpu=mips32
file.c
  -O3 max speed | -Os size | -S emit
asm | -target triple
```

Compiler output uses $fp/$gp, .cfi directives, delay-slot nops, loop unrolling, strength reduction (×4 → sll), keeps vars in regs → fewer lw/sw than hand code; MARS can't assemble it directly. No MIPS stdlib with clang.

E · MIPS SINGLE-CYCLE CPU

ALU (F2:0) — F2 inverts B, F1:0 picks output

F2:0	Op	F2:0	Op
000	A AND B	100	A AND ~B
001	A OR B	101	A OR ~B
010	A + B	110	A − B (~B + 1, Cin=F2)
011	unused	111	SLT

B' = F2 ? ~B : B; Sum = A + B' + F2. Mux F1:0: 00→AND, 01→OR, 10→Sum, 11→ZeroExtend(Sum[N−1]) (SLT = sign of A−B).
Zero = NOR(all Y bits).

ALU decoder

ALUOp	Funct	ALUControl
00	X	010 add (lw/sw/addi)
X1 (01)	X	110 sub (beq)
1X	100000 add	010
1X	100010 sub	110
1X	100100 and	000
1X	100101 or	001
1X	101010 slt	111

Main decoder

Instr	Op	RW	RD	AS	Br	MW	M2R	ALUOp	J
R	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

RW=RegWrite, RD=RegDst, AS=ALUSrc, Br=Branch, MW=MemWrite, M2R=MemtoReg. **PCSrc = Branch · Zero**.

Datapath (signal flow)

```
PC → IMem(A) → Instr
Instr[25:21]=rs → RF.A1 → RD1 = SrcA
Instr[20:16]=rt → RF.A2 → RD2
WriteReg A3 = RegDst ? Instr[15:11]
(rd) : Instr[20:16](rt)
SignImm = SignExt(Instr[15:0])
SrcB = ALUSrc ? SignImm : RD2
ALUResult = ALU(SrcA,SrcB,ALUControl)
; Zero
DMem: A=ALUResult, WD=RD2, WE=MemWrite
Result = MemtoReg ? ReadData :
ALUResult → RF.WD3 (WE3=RegWrite)
PCPlus4 = PC+4
PCBranch = PCPlus4 + (SignImm<<2)
PC' = Jump ?
{PCPlus4[31:28],Instr[25:0],00}
        : (PCSrc ? PCBranch :
PCPlus4)
```

Logisim: PC = 32-bit register; IMem = ROM loaded with memfile.dat ("v2.0 raw" header), addressed by PC[x:2] (word index); DMem = RAM, addr = ALUResult[x:2].

Test program (memfile.dat) — success: writes 7 to addr 84

Ad	Assembly	Effect	Machine
00	main: addi $2,$0,5	$2=5	20020005
04	addi $3,$0,12	$3=12	2003000c
08	addi $7,$3,-9	$7=3	2067fff7
0c	or $4,$7,$2	$4=3|5=7	00e22025
10	and $5,$3,$4	$5=12&7=4	00642824
14	add $5,$5,$4	$5=4+7=11	00a42820
18	beq $5,$7,end	not taken	10a7000a
1c	slt $4,$3,$4	$4=12<7=0	0064202a
20	beq $4,$0,around	taken	10800001
24	addi $5,$0,0	shouldn't run	20050000
28	around: slt $4,$7,$2	$4=3<5=1	00e2202a
2c	add $7,$4,$5	$7=1+11=12	00853820
30	sub $7,$7,$2	$7=12−5=7	00e23822
34	sw $7,68($3)	[80]=7	ac670044
38	lw $2,80($0)	$2=[80]=7	8c020050
3c	j end	taken	08000011
40	addi $2,$0,1	shouldn't run	20020001
44	end: sw $2,84($0)	[84]=7	ac020054

F · 8086 / XPO-86

8086 registers

AX (AH/AL)	accumulator; MUL/DIV, I/O
BX (BH/BL)	base (addressing [BX])
CX (CH/CL)	count (LOOP, shifts by CL)
DX (DH/DL)	data; MUL high word; I/O port
SI / DI	source / dest index
SP / BP	stack ptr / base ptr (uses SS)
CS DS SS ES	code/data/stack/extra segments
IP	instruction ptr (CS:IP)
FLAGS	CF PF AF ZF SF TF IF DF OF

Physical addr = Segment×10h + Offset (20-bit, 1 MB). 1000:0100 → 10100h; 2000:0050 → 20050h.

8086: 16-bit CPU, 20 addr lines, 40-pin, 5/8/10 MHz, min mode (single CPU) / max mode (multiprocessor), BIU + EU with 6-byte prefetch queue.

MUL src (unsigned): 8-bit AL×src→AX; 16-bit AX×src→**DX:AX**. IMUL signed. **DIV src**: 16-bit DX:AX/src → AX quot, DX rem; 8-bit: AX/src → AL quot, AH rem. ADC = add with carry, LOOP = CX−−, jump if CX≠0. JZ/JNZ/JC/JNC/JMP, CMP, INC/DEC. **INT A5** = return to XPO monitor.

XPO-86 kit components

P8086-2 CPU · **KM62256** 32K×8 SRAM (no refresh, faster/costlier than DRAM) · **8255 (D82655AC)** programmable peripheral interface (I/O ports) · **8253/8254** 16-bit programmable timer/counter (timer=regular clock, counter=events) · **74LS04** hex inverter (6 NOTs) · EPROM monitor, keyboard, LCD.

XPO-86 commands (default seg 1000, offset 0100)

A	Assemble: A / A SSSS:DDDD. No labels/symbols, hex addrs only.
C	Compare memory: C 1000 10FF 2000
D	Display 128 B hex dump: D 1000:0200 / D 1000:0200 020F
E	Examine/modify byte: E 1000:2000; space=next, '-'=prev
R	Show registers; R DS → edit DS
U	Unassemble: U 1000:0 / U 1000:0 0FFF
G	Go/execute
S	Substitute → memory / register view

Run: E → EXPAND? Enter → A → type lines (1000:0100 MOV AX,5555 …) end with INT A5 → Enter on blank → Q → Enter. **Exec:** G → Enter (Burst) → SRC-SEGM 1000 → ADDR 0100 → Enter. **Check:** S → Enter → ↓↓ register → Enter → name. Esc = command mode; Reset = registers lost.

ALPs (data at DS=1000)

```
; Q2: 16×16 multiply → 32-bit result
MOV AX,[2000]
MOV BX,[2002]
MUL BX          ; DX:AX = AX*BX
MOV [2004],AX   ; low word
MOV [2006],DX   ; high word
INT A5
; Q3: add 3 numbers
MOV AX,[2000]
ADD AX,[2002]
ADD AX,[2004]
MOV [2006],AX
INT A5
; Demo: MOV AX,5555 / MOV BX,4444 /
ADD AX,BX → AX=9999
; Q4: add N numbers (N at 2000, data
from 2002)
     MOV SI,2000
     MOV CX,[SI]     ; CX = N
     MOV AX,0000
     MOV DX,0000     ; carries
     ADD SI,2
L:   ADD AX,[SI]     ; ← note this
addr, e.g. 0110
     ADC DX,0000
     ADD SI,2
     LOOP L          ; type the hex
addr of L
     MOV [3000],AX
     MOV [3002],DX
     INT A5
```

G · DE1-SoC MEMORY-MAPPED I/O

Address map

Device	Address	Bits
LEDR	FF200000	9:0 (LED9=0x200, all=0x3FF)
HEX3–0	FF200020	HEX0 6:0, HEX1 14:8, HEX2 22:16, HEX3 30:24
HEX5–4	FF200030	HEX4 6:0, HEX5 14:8
SW	FF200040	9:0
KEY	FF200050	3:0, reads 1 while pressed (+0xC edge-capture)
Pixel buf	Nios 08000000 · **ARM C8000000**	320×240, 16-bit
Char buf	Nios 09000000 · **ARM C9000000**	80×60, 8-bit

CPUlator: cpulator.01xz.net/?sys=arm-de1soc | ?sys=nios-de1soc

7-segment codes (bit6..0 = g f e d c b a, 1 = on)

0	3F	1	06	2	5B	3	4F	4	66	5	6D	6	7D	7	07
8	7F	9	6F	A	77	b	7C	C	39	d	5E	E	79	F	71

ARM ↔ Nios II

Action	ARMv7	Nios II
load addr	LDR R8,=0xFF200000	movia r8,0xFF200000
small const	MOV R9,#0x200	movi r9,0x200
write I/O	STR R9,[R8]	stwio r9,0(r8)
read I/O	LDR R0,[R7]	ldwio r2,0(r7)
byte/half	LDRB/STRB/STRH	ldbu/stbio/sthio
test bit0	TST R0,#1 ; BEQ L	andi r3,r2,1 ; beq r3,zero,L
compare	CMP R0,R1 ; BGE L	bge r2,r3,L
decrement	SUBS R10,R10,#1 ; BNE L	subi r10,r10,1 ; bne r10,zero,L
shift	LSL R0,R0,#1 / LSR	slli r2,r2,1 / srli
call / ret	BL f / BX LR	call f / ret
jump	B L	br L
const sym	.equ LED,0xFF200000	.equ (same)

ARM: R13=SP, R14=LR, R15=PC; flags NZCV only set with S suffix / CMP / TST; cond suffix EQ NE GT LT GE LE (e.g. MOVEQ). PUSH {R4,LR} / POP {R4,PC}. Entry `.global _start`. Nios: r0=zero, r31=ra, r27=sp.

ARM templates

```
        .equ LED, 0xFF200000
        .equ SW,  0xFF200040
        .global _start
_start: LDR R2,=LED
        LDR R1,=SW
@ Task1: LED0 only → MOV R0,#1; LED9 →
#0x200; all → LDR R0,=0x3FF; STR R0,
[R2]
@ Task2 blink LED0+LED9 together /
alternately
B1:     LDR R0,=0x201      @
alternate: 0x001 then 0x200
        STR R0,[R2]
        BL  DELAY
        MOV R0,#0
        STR R0,[R2]
        BL  DELAY
        B   B1
@ Task3: SW → LED
COPY:   LDR R0,[R1]
        STR R0,[R2]
        B   COPY
@ Task4: blink LED9 while SW0, SW1 =
speed
T4:     LDR R0,[R1]
        TST R0,#1
        MOVEQ R3,#0
        STREQ R3,[R2]
        BEQ T4
        LDR R10,=1000000   @ slow
        TST R0,#2
        LDRNE R10,=250000  @ fast
        EOR R4,R4,#0x200   @ toggle
LED9
        STR R4,[R2]
W:      SUBS R10,R10,#1
        BNE W
        B   T4
@ Task5: running LED, SW0 picks
direction
        MOV R0,#1
RUN:    STR R0,[R2]
        BL  DELAY
        LDR R3,[R1]
        TST R3,#1
        BNE RIGHT
        LSL R0,R0,#1       @ LED0 →
LED9
        CMP R0,#0x400
        MOVEQ R0,#1
        B   RUN
RIGHT:  LSR R0,R0,#1       @ LED9 →
LED0
        CMP R0,#0
        MOVEQ R0,#0x200
        B   RUN
DELAY:  LDR R10,=1000000
DL:     SUBS R10,R10,#1
        BNE DL
        BX  LR
```

```
@ Task6/7/8: HEX0 (R7=0xFF200020,
R6=KEY, R8=SEG)
        LDR R6,=0xFF200050
        LDR R7,=0xFF200020
        LDR R8,=SEG
@ Task6: show 'A' → MOV R0,#0x77; STR
R0,[R7]
@ 0..F: loop R5=0..15: LDRB R0,
[R8,R5]; STR; BL DELAY
@ Task7: SW3–0 → HEX0
T7:     LDR R0,[R1]
        AND R0,R0,#0xF
        LDRB R0,[R8,R0]
        STR R0,[R7]
        B   T7
@ Task8: KEY0 ++, KEY1 reset (one
count per press)
        MOV R5,#0
SHOW:   LDRB R0,[R8,R5]
        STR R0,[R7]
WP:     LDR R0,[R6]        @ wait for
press
        CMP R0,#0
        BEQ WP
        MOV R1,R0          @ remember
key
WR:     LDR R0,[R6]        @ wait for
release
        CMP R0,#0
        BNE WR
        TST R1,#2          @ KEY1?
        MOVNE R5,#0
        BNE SHOW
        ADD R5,R5,#1
        AND R5,R5,#0xF     @ 15 → 0
        B   SHOW
SEG: .byte
0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x0
7
     .byte
0x7F,0x6F,0x77,0x7C,0x39,0x5E,0x79,0x7
1
```

```
# Nios II blink LED9
_start: movia r8,0xFF200000
blink:  movi  r9,0x200
        stwio r9,0(r8)
        call  delay
        stwio zero,0(r8)
        call  delay
        br    blink
delay:  movia r10,1000000
dl:     subi  r10,r10,1
        bne   r10,zero,dl
        ret
```

VGA (Lab 14)

Pixel addr = base + 1024·y + 2·x (y≪10 | x≪1) **Char addr = base + 128·row + col** (row≪7 | col)

Screen 320×240 (x 0..319, y 0..239); text 80×60. RGB565: **R 5 | G 6 | B 5**.

RED F800	GREEN 07E0	BLUE 001F
WHITE FFFF	BLACK 0000	YELLOW FFE0
CYAN 07FF	MAGENTA F81F	GREY 8410

Worked	Address (Nios bases)
pixel (4,1)	1024+8=1032 → 08000408
pixel (20,10)	10240+40=10280 → 08002828
char (4,1)	128+4=132 → 09000084
char (15,22)	2816+15=2831 → 09000B0F
diag (319−i,i)	starts (319,0), ends (80,239)

```
#define PIXEL_BASE  0x08000000u   /*
ARM: 0xC8000000 */
#define CHAR_BASE   0x09000000u   /*
ARM: 0xC9000000 */
#define SWITCH_BASE 0xFF200040u
void put_pixel(int x,int y,uint16_t c)
{
  *(volatile uint16_t*)
(PIXEL_BASE+1024u*y+2u*x)=c; }
void put_char(int col,int row,char v){
  *(volatile uint8_t*)
(CHAR_BASE+128u*row+col)=v; }
/* switch demo: highest switch wins */
uint32_t s=*(volatile
uint32_t*)SWITCH_BASE & 7u;
c = (s&4)?BLUE:(s&2)?GREEN:(s&1)?
RED:BLACK;
if(c!=prev){ fill_rect(0,0,320,240,c);
prev=c; }
/* bands by x: x<107 RED, x<214 GREEN,
else BLUE */
/* key edge in C */
while(!(*KEYS&1)); while(*KEYS&1);
count++;
```

volatile = force real read/write each time (I/O can change outside the program). Redraw only on change avoids flicker. Nios asm: `sthio r4,1032(r2)` pixel, `stbio r5,132(r3)` char. ARM_game.c: VGA 0xC8000000, KEY0 jump, KEY1 day/night, score on HEX3-0.