Real-Life RISC

MIPS Architecture

RISC Instruction Set Architecture (ISA)

Defines registers + instructions

MIPS cores

Define device-dependent implementation details

Pipeline organization, I/O organization, control registers, ...

MIPS32

32-bit RISC ISA

DLX = simplified MIPS32

MIPS64

64-bit RISC ISA = 64-bit integers / 64-bit addresses

Binary compatible with MIPS32

Applications

Typically licensed to OEMs

Design implemented in embedded systems

MIPS-based PCs used in China

MIPS32 ISA — 1

Registers

32-bit integer registers

`$0, $1, ... , $31`

`Regs[$0] = 0 (read-only)`

32-bit FP registers

`F0, F1, ... , F31`

Special registers

`HI, LO`

64-bit result of integer multiply

Quotient + remainder result of integer divide

Instruction formats

Type	32 26	25 21	20 16	15 0		
	6	5	5	5	5	6
R	opcode	s	t	d	a	function
I	opcode	s	t	immediate		
J	opcode	target				

MIPS32 ISA — 2

Coprocessors

Logical extensions of basic MIPS ISA

Accessed via coprocessor read / write instructions

CP0

System Control Coprocessor — on CPU

Supports virtual memory system and exception handling

- Translates virtual addresses into physical addresses
- Controls cache subsystem
- Handles switches between kernel / supervisor/ user states
- Manages exceptions / diagnostic control / error recovery

CP1

Interface to FPU

CP2

Available for device-specific implementations

CP3

Interface to FPU on MIPS64 and newer MIPS32

MIPS32 Instruction Set — 3

A few MIPS instructions not in DLX

Coprocessor Load / Store	`LWCz t, imm(reg)`	Load Word to Coprocessor_z, z = 1 or 2
	`SWCz imm(reg), t`	Store Word from Coprocessor_z, z = 1 or 2
Test+Set	`SLTI t, s, imm`	Set on Less Than Immediate
Shift	`ROTR`	Rotate Word Right
	`SLL / SRA`	Shift Word Left Logical / Arithmetic
Multiply	`MUL d, s, t`	Multiply to general register
	`MULT s, t`	Multiply to HI/LO (64-bit output)
	`MADD s, t`	Multiply and add result to HI/LO
Extract	`EXT t, s, pos, size`	`t ← substr(s,pos=a,size=d)`
Branch	`BGTZ / BGEZ`	Branch greater / greater or equal zero
	`BLTZ / BLEZ`	Branch less / less or equal zero
	`BNE`	Branch not equal
Synchronize	`SYNC`	Critical section for shared memory
System	`SYSCALL`	System Call
Trap	`TEQ / TGE / TNE`	Trap if equal / greater or equal / not equal
Cache	`PREF`	Prefetch

MIPS64 ISA

Registers

64-bit integer registers

`$0, $1, ... , $31`

`Regs[$0] = 0 (read-only)`

32-bit FP registers on 32-bit FPU

64-bit FP registers on 64-bit FPU

`F0, F1, ... , F31`

Special registers

`HI, LO`

128-bit result of integer multiply

Quotient + remainder result of integer divide

Instruction formats

32-bit instruction length — binary compatible with MIPS32

MIPS32/64 instructions act on lower 32-bits in registers

MIPS64_double instructions act on full 64-bits in registers

Memory address = 64-bit pointer (register) + 16-bit immediate

WinMIPS

Cross-compiler

Develop applications for MIPS in C / C++ on Windows platforms

Extensive documentation

Tools

GNU GCC compiler and assembler

GNU-Utils for compiler/linker

Download

https://www.ocf.berkeley.edu/~davelai/al/labs/lab10/

Requires cygwin1.dll

Convert C code to assembly code

```
cc1.exe filename.c -o filename.asm
```


MIPS Compilation 1-1

C source

```
main()
{
      int x = 0;
      while (x < 10){
         x++;
      }
}
```

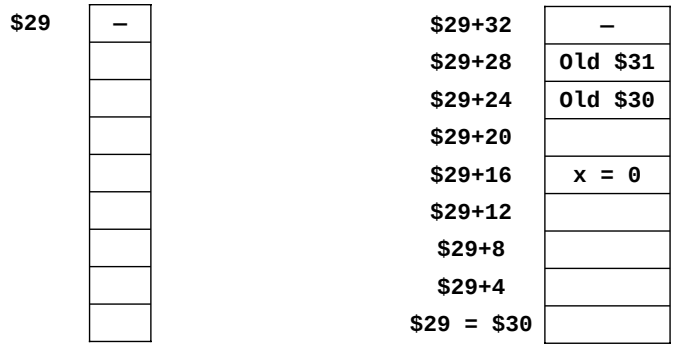
Assembly output of compiler

```
main:
      sub $29,$29,32
      sw $31,28($29)
      sw $30,24($29)
      addi $30,$29,$0
      jal __main
      sw $0,16($30)
$L2:
      lw $2,16($30)
      slt $2,$2,10
      bne $2,$0,$L4
      j $L3
$L4:
      lw $2,16($30)
      addu $2,$2,1
      sw $2,16($30)
      j $L2
$L3:
      addi $29,$30,$0
      lw $31,28($29)
      lw $30,24($29)
      addu $29,$29,32
      j $31
```

MIPS Compilation 1-2

Building data frame

```
sub $29,$29,32          ; $29 used as stack pointer
sw $31,28($29)          ; save $31 = old return pointer
sw $30,24($29)          ; save $30 = old frame pointer
addi $30,$29,$0         ; set $30 = $29
jal __main              ; OS function call to set up C
                        ; saves return pointer in $31
sw $0,16($30)           ; store x = 0 in data frame
```



MIPS Compilation 1-3

Executing loop

```
$L2:
     lw $2,16($30)          ; $2 ← x
     slt $2,$2,10           ; if ($2 < 10) $2 ← 1, else $2 ← 0
     bne $2,$0,$L4          ; if ($2 != 1) jump to $L4
     j $L3                  ; jump to $L3 (exit loop)
$L4:
     lw $2,16($30)          ; $2 ← x
     addu $2,$2,1           ; $2 ← $2 + 1
     sw $2,16($30)          ; x ← $2
     j $L2                  ; jump to $L2 (continue loop)
$L3:
     addi $29,$30,$0        ; restore $29
     lw $31,28($29)         ; restore $31 (old return pointer)
     lw $30,24($29)         ; restore $30 (old frame pointer)
     addu $29,$29,32        ; restore original $29
     j $31                  ; return to calling point
```


MIPS Compilation 2-1

C source

```
main()
{
  int x , y;
  for (x = 0 ; x < 10 ; x++){
    y = x + 4;
  }
}
```

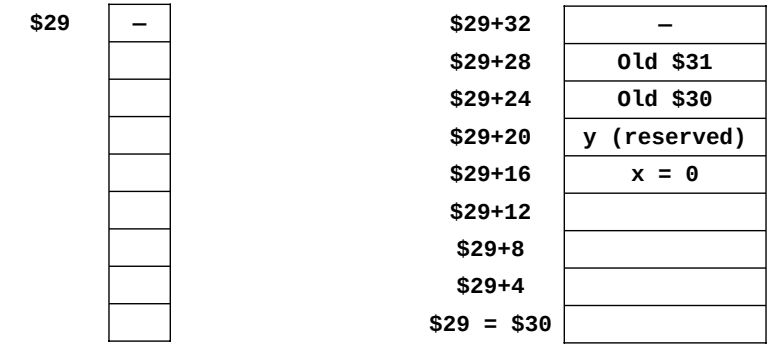
Assembly output of compiler

```
main:
      sub $29,$29,32
      sw $31,28($29)
      sw $30,24($29)
      addi $30,$29,$0
      jal__main
      sw $0,16($30)
$L2:
      lw $2,16($30)
      slt $2,$2,10
      bne $2,$0,$L5
      j $L3
$L4:
      lw $2,16($30)
      addu $2,$2,4
      sw $2,20($30)
      lw $2,16($30)
      add $2,$2,1
      sw $2,16($30)
      j $L2
$L3:
      addi $29,$30,$0
      lw $31,28($29)
      lw $30,24($29)
      add $29,$29,32
      j $31
```


MIPS Compilation 2-2

Building data frame

```
sub $29,$29,32          ; $29 used as stack pointer
sw $31,28($29)          ; save $31 = old return pointer
sw $30,24($29)          ; save $30 = old frame pointer
addi $30,$29,$0         ; set $30 = $29
jal __main              ; OS function call to set up C
                        ; saves return pointer in $31
sw $0,16($30)           ; store x = 0 in data frame
```



MIPS Compilation 3-3

Executing loop

```
$L2:
      lw $2,16($30)              ; $2 ← x
      slt $2,$2,10               ; if ($2 < 10) $2 ← 1, else $2 ← 0
      bne $2,$0,$L4              ; if ($2 != 1) jump to $L4
      j $L3                      ; jump to $L3 (exit loop)
$L4:
      lw $2,16($30)              ; $2 ← x
      addu $2,$2,4               ; $2 ←$2 + 4
      sw $2,20($30)              ; y ←$2
      lw $2,16($30)              ; $2 ← x
      add $2,$2,1                ; $2 ← $2 + 1
      sw $2,16($30)              ; x ← $2
      j $L2                      ; jump to $L2 (continue loop)
$L3:
      addi $29,$30,$0            ; restore $29
      lw $31,28($29)             ; restore $31 (old return pointer)
      lw $30,24($29)             ; restore $30 (old frame pointer)
      addu $29,$29,32            ; restore original $29
      j $31                      ; return to calling point
```


ARM Overview

Microprocessor and microcontroller for embedded systems

- **A**dvanced **R**ISC **M**achine developed by ARM Limited
- ARM Ltd primarily licenses ISA implementations to developers
- Most widely used 32-bit RISC ISA
  - Over 50 billion ARM processors used in phones, games, peripherals
  - 98 percent of mobile phones use ARM



RISC Architectural Features

Data Types

- Byte 8 bits
- Halfword 16 bits (in ARMv4 and higher)
- Word 32 bits

Standard RISC

- Load/store architecture
- Large uniform register file
- Simple addressing modes
- Uniform and fixed-length instruction fields
- Scalar in-order pipeline

Additional ARM architectural features

- Shift + ALU operations
- Auto-increment / auto-decrement addressing modes for loops
- Load and Store Multiple instructions
- Conditional execution of most instructions
  - Cancel instructions on certain condition flags
  - Replaces control hazards in forward jumps

ARM Versions

Architecture Version	Processor Family
ARMv1	ARM1
ARMv2	ARM2
	ARM3
ARMv3	ARM6
	ARM7
ARMv4	ARM7TDMI
	StrongARM
	ARM8
	ARM9TDMI
ARMv5TE ARMv5TEJ	ARM9E
	ARM10E
	XScale
ARMv6	ARM11
ARMv7	Cortex
ARMv8	Cortex

Extension Features	
J	Jazelle (Java)
T	Thumb
D	Debugger
M	Multiplier (64-bit result)
I	ICE
E	DSP Enhancement (implies TDMI)

Seven Operating Modes

User	Normal (non-privileged) program execution mode No access to protected resources
FIQ	Supports high speed data transfer or DMA process Entered on high priority (fast) interrupt
IRQ	General purpose interrupt handling Entered on low priority (normal) interrupt
Supervisor	Protected mode for operating system Entered on reset and on Software Interrupt
Abort	Implements virtual memory and/or memory protection Handles memory access violations
Undef	Supports software emulation of hardware coprocessors
System	Runs privileged operating system tasks (ARMv4 and above) Accesses user mode registers

Registers

32-bit general purpose registers

- 16 architectural registers
  - r0, ... , r15
  - r11 — FP = Frame Pointer
  - r12 — IP = intra-procedure-call scratch register
  - r13 — SP = Stack Pointer used by push/pop instructions
  - r14 — LR = Link Register used to return from function calls
  - r15 — PC = Program Counter
- 31 physical registers
  - Multiple copies of r8, ... , r14
  - Each copy accessible in specific operating mode

32-bit status registers

- Current Program Status Register (CPSR) visible in all modes
  - 32 bits wide
- 5 Saved Program Status Registers (SPSR)
  - Privileged modes copy previous CPSR

Modes and Visible Registers

User	FIQ	IRQ	Supervisor	Abort	Undef
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8	r8	r8	r8	r8
r9	r9	r9	r9	r9	r9
r10	r10	r10	r10	r10	r10
r11 (FP)	r11 (FP)	r11 (FP)	r11 (FP)	r11 (FP)	r11 (FP)
r12 (IP)	r12 (IP)	r12 (IP)	r12 (IP)	r12 (IP)	r12 (IP)
r13 (SP)	r13 (SP)	r13 (SP)	r13 (SP)	r13 (SP)	r13 (SP)
r14 (LR)	r14 (LR)	r14 (LR)	r14 (LR)	r14 (LR)	r14 (LR)
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR	SPSR	SPSR	SPSR	SPSR

Instruction Sets

ARM

- 32-bit instructions
- Aligned on 32-bit boundaries
  - Lowest 2 bits of PC (r15) always 0

Thumb

- 16-bit instructions 1-to-1 mapped to 32-bit instructions
- Shortened versions with restricted options and implicit operands
  - Example — **`add dest, src1, src2`** becomes **`add dest, src`**
- Aligned on 16-bit boundaries
  - Lowest bit of PC (r15) always 0
- Set by T = 1 in CPSR

Jazelle

- Executes Java bytecode directly
- ARM reads 4 8-bit instructions per instruction fetch
- Set by J = 1 in CPSR

Conditional Execution

Flag set suffix S

ALU instructions set CPSR flags

- N — Negative
- Z — Zero
- C — Carry
- V — oVerflow

Conditional execution suffix

Execute instructions if flag combination is true

Example — operate-compare-if-else

Usual Execution	Conditional Execution
`SUB r3, r3, #1`	`SUBS r3, r3, #1`
`CMP r3, #0`	
`BEQ L1 ; branch on 0`	
`ADD r0, r1, r2 ; skip if r3 = 0`	`ADDNE r0, r1, r2`
`B L2 ; jump to L2`	
`L1: SUB r0, r1, r2 ; skip if r3 != 0`	`SUBEQ r0, r1, r2`
`L2: ADD r4, r5, r6`	`ADD r4, r5, r6`

Basic Instructions

Transfer

MOV — Move register-to-register or immediate-to-register

MVN — Move Not

LDR, STR — load / store

Branch

B — Add sign-extended 24-bit signed immediate to PC (r15)

BL — Branch and store PC+4 in link register (r14)

Conditional branch — conditional execution of **B** or **BL**

ALU

ADD	Add	**MLA**	Multiply and ADD
ADC	Add with Carry	**ORR**	Logical OR
AND	Logical AND	**SUB**	Subtract
BIC	Logical Bit Clear	**SBC**	Subtract with Carry
CMP	Compare	**RSB**	Reverse Subtract
CMN	Compare Negative	**RSC**	Reverse Subtract with Carry
EOR	Logical XOR	**TST**	Test
MUL	Multiply	**TEQ**	Test Equivalence

Shift and Rotate

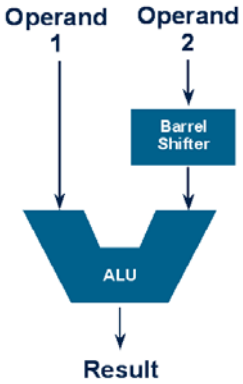
Legal ALU instruction operands

32-bit source / destination register contents

Sign-extended 12-bit immediate

Shifted-operand

- Shifted / rotated 32-bit source register contents
- Number of shifts set by 8-bit immediate



Shifts

LSL — Logical Left Shift (unsigned)

ASR — Arithmetic Right Shift

LSR — Logical Shift Right

Rotates

ROR — Rotate Right

RRX — Rotate Right Extended (CF into MSB)

VFP Extension

No FP operations in basic instruction set

Not needed in simple embedded applications

VFP implements FP ISA extension as optional coprocessor

Since ARM10

Vector Floating Point (VFP)

Single precision and double precision floating point computation

- ANSI/IEEE Std 754 compliant

Single Instruction Multiple Data (SIMD) FP unit

- One FP operation performed in parallel on 256-bit vector
  - 8 single-precision (4-byte) FP numbers
  - 4 double-precision (8-byte) FP numbers
- Accesses 32 single precision FP registers (32-bit width)

VFPv3

Operates on 8 double-precision (8-byte) FP numbers (512-bit vector)

VFP Instructions

Transfer

Load / store FP values into registers from memory
Transfer / copy 32-bit values between VFP and ARM GP registers
Conversions between float, double, unsigned / signed integers

FPU

Add
Subtract
Multiply
Divide
Square root
Combined multiply-accumulate
Compare FP values in registers

VFPv3

Store FP constant in register

DSP Enhancements

Digital Signal Processing

Operations on sampled-digitized-encoded analog information
Typical applications
- Audio / video, speech processing, modems, medical instruments

Typical algorithms
- D / A, normalization, correlation, convolution, FFT, encoding / decoding
- Real time control

Practical applications

GSM-AMR (Adaptive Multi-Rate) speech codec in 3G GSM phones
Servo motor control (HDD/DVD)
Audio encode/decode (MP3, AAC, WMA)
MPEG4 decode
Voice and handwriting recognition

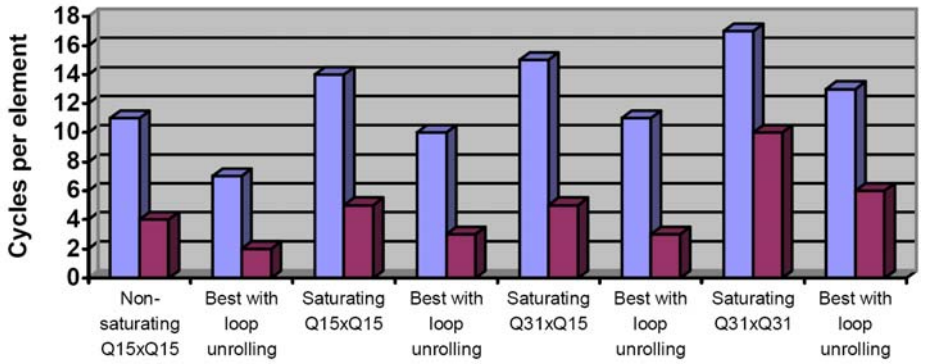
DSP Instructions

Instruction	Operation	Purpose
`SMLAxy{cond}`	`16 x 16 + 32 → 32`	Signed MAC
`SMLAWy{cond}`	`32 x 16 + 32 → 32`	Signed MAC wide
`SMLALxy{cond}`	`16 x 16 + 64 → 64`	Signed MAC long
`SMULxy{cond}`	`16 x 16 → 32`	Signed multiply
`SMULWy{cond}`	`16 x 32 → 32`	Signed multiply long
`QADD Rd, Rm, Rs`	`SAT(Rm + Rd)`	Saturating add
`QDADD Rd, Rm, Rs`	`SAT(Rm + SAT(Rs x 2))`	Saturating add double
`QSUB Rd, Rm, Rs`	`SAT(Rm – Rd)`	Saturating subtract
`QDSUB Rd, Rm, Rs`	`SAT(Rm – SAT(Rs x 2))`	Saturating subtract double
`CLZ{cond} Rd, Rm`	`COUNTZ(Rm)`	Count leading zeros

16 — halfword
32 — word
64 — doubleword

MAC	Multiply – Accumulate (**Rd ← R1 x R2 + R3**)
Saturating	Pin overflow result at max or min No modulo arithmetic or report of overflow

Performance Comparisons

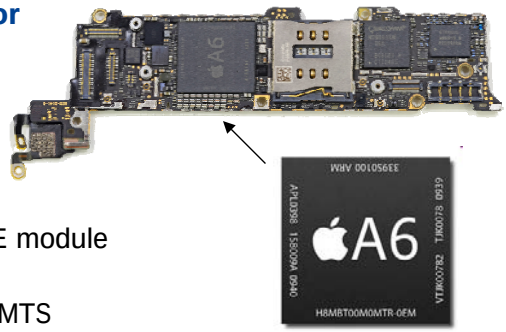


Q15 / Q31 — integer arithmetic techniques used in DSP

Apple iPhone 5 Hardware

Apple A6 Application Processor

- Dual ARMv7 cores + 3 GPU
- Hand-optimized layout

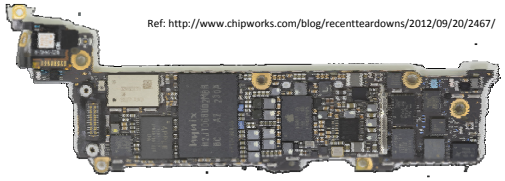


Memory

- Hynix 16 GB Flash

Network processors

- Skyworks GSM / GPRS / EDGE module
- Skyworks CDMA module
- Triquint WCDMA / HSUPA / UMTS
- Qualcomm LTE processor
- Murata WiFi module



Ref: http://www.chipworks.com/blog/recentteardowns/2012/09/20/2467/

Interface controllers

- Apple Power Management IC
- Apple Audio CODEC
- Texas Instruments touch screen controller
- STMicroelectronics 3-axis gyro
- STMicroelectronics 3-axis linear accelerometer

WinARM

Cross-compiler

- Develop applications for ARM in C / C++ on Windows platforms
- Extensive documentation

Tools

- GNU GCC compiler
- GNU-Utils for compiler/linker
- ARM header-files
- Sample applications with source-code

Download

http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects

Convert C code to assembly code

```
arm-elf-gcc -S filename.c -o filename.asm
```


ARM Compilation 1-1

C source

```
main()
{
     int x = 0;
     while (x < 10){
        x++;
     }
}
```

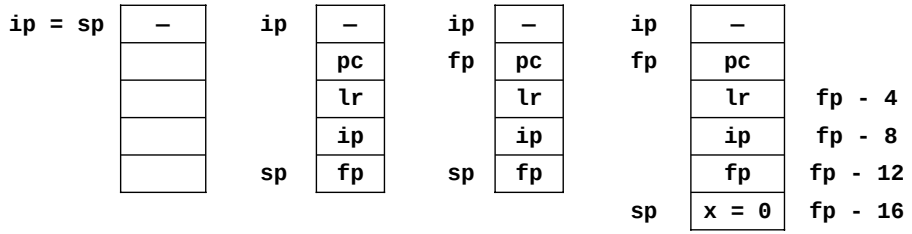
Assembly output of compiler

```
main:
     mov       ip, sp
     stmfd     sp!, {fp, ip, lr, pc}
     sub       fp, ip, #4
     sub       sp, sp, #4
     mov       r3, #0
     str       r3, [fp, #-16]
     b  .L2
.L3:
     ldr       r3, [fp, #-16]
     add       r3, r3, #1
     str       r3, [fp, #-16]
.L2:
     ldr       r3, [fp, #-16]
     cmp       r3, #9
     ble       .L3
     ldmfd     sp, {r3, fp, sp, pc}
```


ARM Compilation 1-2

Building data frame

```
     mov       ip, sp
     stmfd     sp!, {fp, ip, lr, pc}
       ; push fp, ip, lr, pc to stack
     sub       fp, ip, #4
       ; fp = ip – 4
       ; sp = ip – 16 = fp - 12
     sub       sp, sp, #4
       ; sp = fp – 16
     mov       r3, #0
     str       r3, [fp, #-16]
```



ARM Compilation 1-3

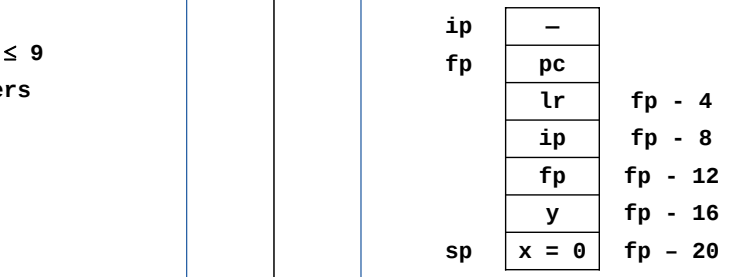
Executing loop

```
       b       .L2                     ; branch to .L2
.L3:
     ldr       r3, [fp, #-16]          ; r3 ← x
     add       r3, r3, #1              ; r3++
     str       r3, [fp, #-16]          ; x ← r3
.L2:
     ldr       r3, [fp, #-16]          ; r3 ← x
     cmp       r3, #9                  ; compare r3 , 9
     ble       .L3                     ; jump .L3 if r3 ≤ 9
     ldmfd     sp, {r3, fp, sp, pc}    ; restore registers
```


ARM Compilation 2-1

C source

```
main()
{
  int x , y;
  for (x = 0 ; x < 10 ; x++){
    y = x + 4;
  }
}
```



Assembly output of compiler

```
main:
     mov    ip, sp
     stmfd  sp!, {fp, ip, lr, pc}
     sub    fp, ip, #4
     sub    sp, sp, #8 ; 2 integers
     mov    r3, #0
     str    r3, [fp, #-20]
     b  .L2
.L3:
     ldr    r3, [fp, #-20]
     add    r3, r3, #4
     str    r3, [fp, #-16]
     ldr    r3, [fp, #-20]
     add    r3, r3, #1
     str    r3, [fp, #-20]
.L2:
     ldr    r3, [fp, #-20]
     cmp    r3, #9
     ble    .L3
     sub    sp, fp, #12
     ldmfd  sp, {fp, sp, pc}
```


ARM Compilation 2-2

Executing loop

```
     b  .L2
.L3:
     ldr    r3, [fp, #-20]          ; r3 ← x
     add    r3, r3, #4              ; r3 ← r3 + 4
     str    r3, [fp, #-16]          ; y ← r3
     ldr    r3, [fp, #-20]          ; r3 ← x
     add    r3, r3, #1              ; r3++
     str    r3, [fp, #-20]          ; x ← r3
.L2:
     ldr    r3, [fp, #-20]          ; r3 ← x
     cmp    r3, #9                  ; compare r3 , 9
     ble    .L3                     ; jump .L3 if r3 ≤ 9
     sub    sp, fp, #12             ; sp ← fp – 12
     ldmfd  sp, {fp, sp, pc}        ; restore registers
```