QXD0133 - Arquitetura e Organização de Computadores II

Universidade Federal do Ceará - Campus Quixadá

Thiago Werlley
thiagowerlley@ufc.br

18 de outubro de 2025

Capítulo 4

Capítulo 4

Thumb Instruction Set

Introdução

- Já vimos como são as instruções do **ARM: têm 32 bits**; elas contêm campos para especificar os operandos de operação, origem e destino; eles especificam se sua execução é baseada em uma condição; etc.

Introdução

- Já vimos como são as instruções do **ARM: têm 32 bits**; elas contêm campos para especificar os operandos de operação, origem e destino; eles especificam se sua execução é baseada em uma condição; etc.
- Este formato também **existe desde o primeiro** processador ARM1.

Introdução

- Já vimos como são as instruções do **ARM: têm 32 bits**; elas contêm campos para especificar os operandos de operação, origem e destino; eles especificam se sua execução é baseada em uma condição; etc.
- Este formato também **existe desde o primeiro** processador ARM1.
- No início dos anos 80, muitos processadores tinham instruções de 8 ou 16 bits, então a questão acabou sendo levantada:

Introdução

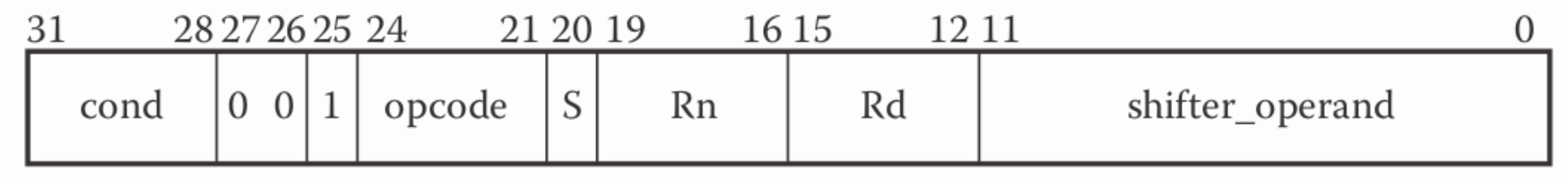
- Já vimos como são as instruções do **ARM: têm 32 bits**; elas contêm campos para especificar os operandos de operação, origem e destino; eles especificam se sua execução é baseada em uma condição; etc.
- Este formato também **existe desde o primeiro** processador ARM1.
- No início dos anos 80, muitos processadores tinham instruções de 8 ou 16 bits, então a questão acabou sendo levantada:
  - **você pode compactar uma instrução de 32 bits**, mantendo seus aprimoramentos e recursos de densidade de código, para aproveitar a memória barata de 16 bits e melhorar ainda mais a densidade do código se você tiver memória de 32 bits?

Introdução

- **Reduzir o tamanho das instruções** existentes pode ser feito examinando os operandos e os campos de bits necessários, e talvez com uma instrução mais curta.

Introdução

- **Reduzir o tamanho das instruções** existentes pode ser feito examinando os operandos e os campos de bits necessários, e talvez com uma instrução mais curta.
- No **padrão de 32 bits** para o **ADD**, por exemplo, os argumentos necessários incluem um registro de destino **Rd**, um registro de origem **Rn** e um segundo operando que pode ser um valor **imediato** ou um **registro**.



Thumb Mode

- O conjunto de instruções Thumb é composto por instruções de 16 bits, que operam de forma equivalente a um subconjunto de instruções ARM de 32 bits.

Thumb Mode

- O conjunto de instruções Thumb é composto por instruções de 16 bits, que operam de forma equivalente a um subconjunto de instruções ARM de 32 bits.
  - Códigos no modo Thumb são, em média, 30% menores que no modo ARM.

Introdução

- Uma instrução simples que adiciona 1 ao registro r2 seria:

```
ADD r2, r2, #1
```

- Essa instrução poderia ser facilmente compactada.
- Principalmente porque o registro de destino é o mesmo que o único registro de origem.
- O outro argumento da adição é 1, um número pequeno o suficiente para caber em um campo de 8 bits.
- Se aplicarmos algumas restrições ao novo conjunto de instruções, a mesma operação poderá ser feita usando apenas um código de operação de 16 bits

```
ADD r2, #1
```

Introdução

- Os campos do operando **mudaram de 4 para 3 bits**, o que significa que os registros permitidos variam de **r0 a r7**, conhecidos como registros baixos.
- Para acessar os outros registros, conhecidos como registros altos, existem instruções separadas.
- A primeira versão desse conjunto de instruções alternativo é chamado Thumb, também conhecido como **Thumb de 16 bits**.
- Ele tem seu próprio estado no processador.
- As instruções são um subconjunto do conjunto de instruções do ARM, o que significa que nem todas as instruções no ARM estão disponíveis no Thumb.

Thumb Mode

- Exemplo:

ARM code

```
ARMDivide
; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0
loop
        SUBS    r0,r0,r1
        ADDGE   r3,r3,#1
        BGE     loop
        ADD     r2,r0,r1
```

$5 \times 4 = 20$ bytes

Thumb code

```
ThumbDivide
; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0
loop
        ADD     r3,#1
        SUB     r0,r1
        BGE     loop
        SUB     r3,#1
        ADD     r2,r0,r1
```

$6 \times 2 = 12$ bytes

Thumb Mode

- Cada instrução Thumb possui instrução equivalente no modo (estado) ARM.

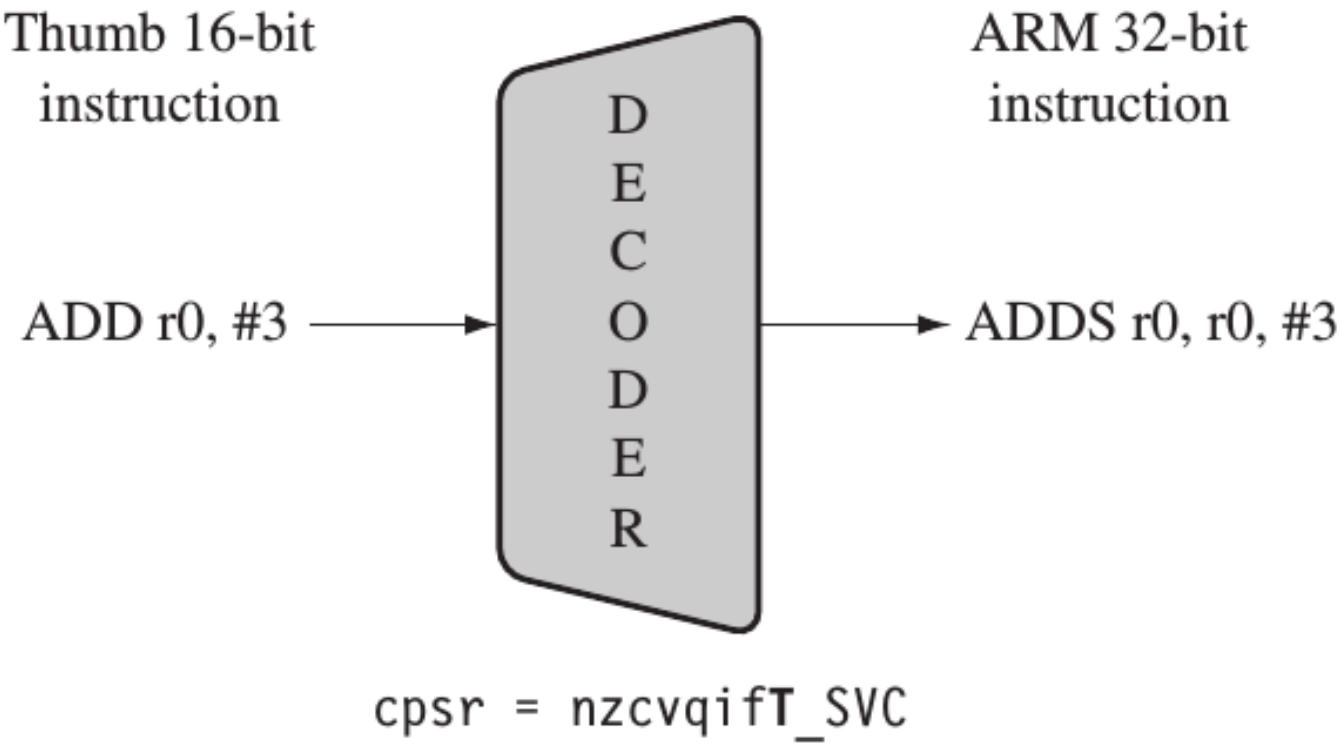
Thumb Mode

- Cada instrução Thumb possui instrução equivalente no modo (estado) ARM.
  - Instruções thumb são decodificadas em instruções ARM.

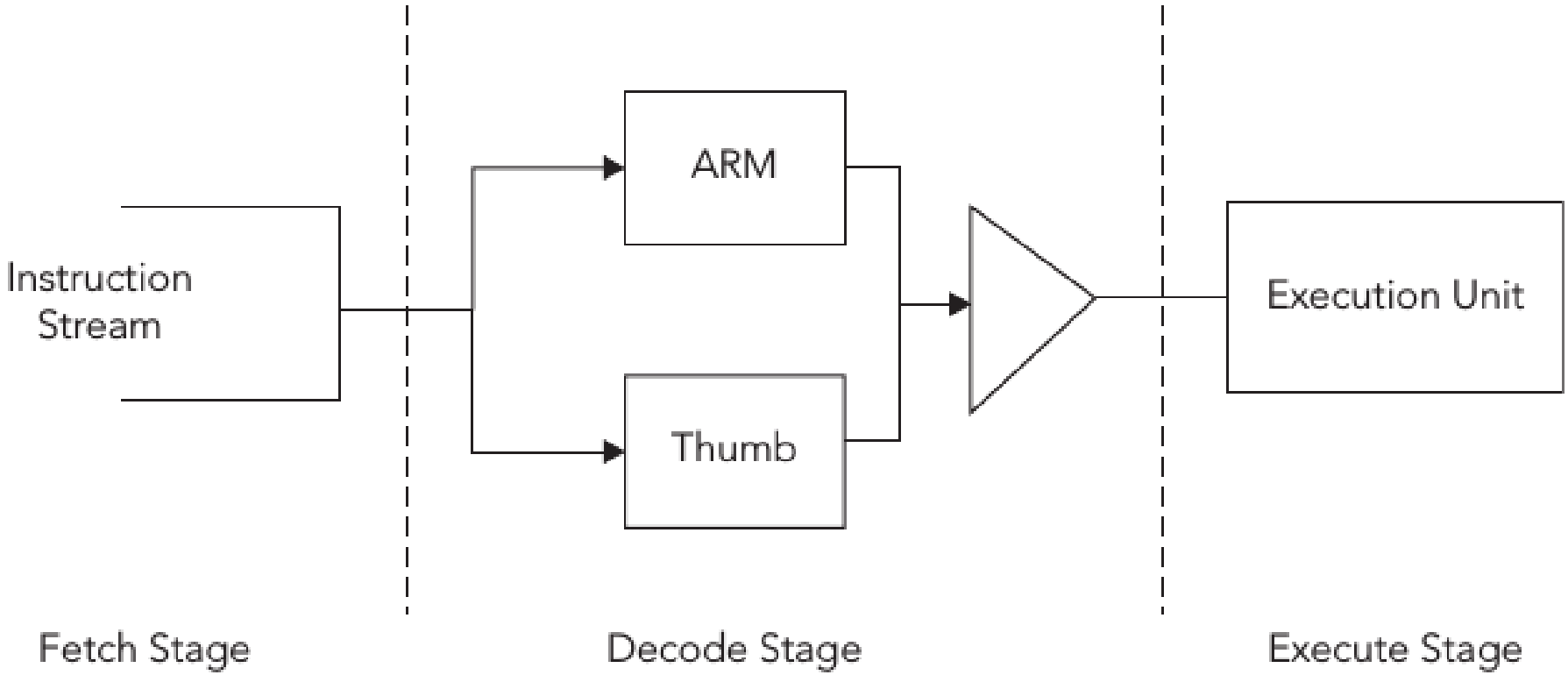
Thumb Mode

- Cada instrução Thumb possui instrução equivalente no modo (estado) ARM.
  - Instruções thumb são decodificadas em instruções ARM.
  - Nem todas as instruções ARM estão disponíveis do modo Thumb.

Decodificação de instruções Thumb



Decodificação de instruções Thumb



Entrando no modo Thumb

```
Syntax: BX    Rm
        BLX   Rm | label
```

BX	Thumb version branch exchange	$pc = Rn$ & 0xfffffffe $T = Rn[0]$
BLX	Thumb version of the branch exchange with link	$lr =$ (instruction address after the BLX) + 1 $pc = label$, $T = 0$ $pc = Rm$ & 0xfffffffe, $T = Rm[0]$

Entrando no modo Thumb

```
; ARM code
        CODE32                          ; word aligned
        LDR     r0, =thumbCode+1        ; +1 to enter Thumb state
        MOV     lr, pc                  ; set the return address
        BX      r0                      ; branch to Thumb code & mode
        ; continue here
; Thumb code
        CODE16                          ; halfword aligned
thumbCode
        ADD   r1, #1
        BX    lr                        ; return to ARM code & state
```

Entrando no modo Thumb

```
; address(thumbCode) = 0x00010000

; cpsr = nzcvqIFt_SVC
; r0 = 0x00000000

0x00009000        LDR r0, =thumbCode+1

; cpsr = nzcvqIFt_SVC
; r0 = 0x00010001

 0x00009008       BX    r0

; cpsr = nzcvqIFT_SVC
; r0 = 0x00010001
; pc = 0x00010000
```

Entrando no modo Thumb

```
        CODE32
        LDR     r0, =thumbRoutine+1     ; enter Thumb state
        BLX     r0                      ; jump to Thumb code
        ; continue here

        CODE16
thumbRoutine
        ADD     r1, #1
        BX      r14                     ; return to ARM code and state
```

Thumb vs. ARM

- Nem todos os registradores estão disponíveis

Registers	Access
r0–r7	fully accessible
r8–r12	only accessible by `MOV`, `ADD`, and `CMP`
r13 sp	limited accessibility
r14 lr	limited accessibility
r15 pc	limited accessibility
cpsr	only indirect access
spsr	no access

Thumb vs. ARM

- Sem acesso direto aos registradores CPSR e SPSR

Thumb vs. ARM

- Sem acesso direto aos registradores CPSR e SPSR
  - Uso das instruções MSR e MRS somente no modo ARM

Thumb vs. ARM

- Sem acesso direto aos registradores CPSR e SPSR
  - Uso das instruções MSR e MRS somente no modo ARM
  - Todas as operações atualizam o status

Thumb vs. ARM

- Instruções separadas para o Barrel Shifter.
  - ASR, LSR, LSL, ROR

```
PRE     r2 = 0x00000002
        r4 = 0x00000001

        LSL    r2, r4

POST    r2 = 0x00000004
        r4 = 0x00000001
```

Thumb vs. ARM - Branch

```
Syntax: B<cond> label
        B label
        BL label
```

B	branch	$pc = label$
BL	branch with link	$pc = label$ $lr =$ (instruction address after the BL) + 1

Thumb vs. ARM - Branch

- Uso de condicionais apenas na instrução Branch (B).

Thumb vs. ARM - Branch

- Uso de condicionais apenas na instrução Branch (B).
  - Se usada alguma condicional (mnemônico de condição) o endereço de destino deve estar na faixa de até 8 bits (de -256 a +254).

Thumb vs. ARM - Branch

- Uso de condicionais apenas na instrução Branch (B).
  - Se usada alguma condicional (mnemônico de condição) o endereço de destino deve estar na faixa de até 8 bits (de -256 a +254).
  - Se não, o endereço pode estar até 11 bits (de -2048 a +2046)

Thumb vs. ARM - Branch

- Uso de condicionais apenas na instrução Branch (B).
  - Se usada alguma condicional (mnemônico de condição) o endereço de destino deve estar na faixa de até 8 bits (de -256 a +254).
  - Se não, o endereço pode estar até 11 bits (de -2048 a +2046)
  - BL tem um range proximado de +/- 4MB (sem condicional)

Data processing instructions

```
Syntax:
      <ADC|ADD|AND|BIC|EOR|MOV|MUL|MVN|NEG|ORR|SBC|SUB>  Rd, Rm
      <ADD|ASR|LSL|LSR|ROR|SUB> Rd, Rn #immediate
      <ADD|MOV|SUB> Rd,#immediate
      <ADD|SUB> Rd,Rn,Rm
       ADD Rd,pc,#immediate
       ADD Rd,sp,#immediate
      <ADD|SUB> sp, #immediate
      <ASR|LSL|LSR|ROR> Rd,Rs
      <CMN|CMP|TST> Rn,Rm
       CMP Rn,#immediate
       MOV Rd,Rn
```

Data processing instructions

- Instruções que não atuam somente nos registradores r0-r7:

Data processing instructions

- Instruções que não atuam somente nos registradores r0-r7:
  - Com exceção de CMP, nenhuma atualiza as flags de status.

```
MOV     Rd,Rn
ADD     Rd,Rm
CMP     Rn,Rm
ADD     sp, #immediate
SUB     sp, #immediate
ADD     Rd,sp,#immediate
ADD     Rd,pc,#immediate
```

Endereçamento com Load/Store

```
Syntax: <LDR|STR>{<B|H>} Rd, [Rn,#immediate]
        LDR{<H|SB|SH>} Rd,[Rn,Rm]
        STR{<B|H>} Rd,[Rn,Rm]
        LDR Rd,[pc,#immediate]
        <LDR|STR> Rd,[sp,#immediate]
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*
LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend(mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend(mem16[address])*

Endereçamento com Load/Store

Addressing modes.

Type	Syntax
Load/store register	[Rn, Rm]
Base register + offset	[Rn, #immediate]
Relative	[pc|sp, #immediate]

```
PRE     mem32[0x90000] = 0x00000001
        mem32[0x90004] = 0x00000002
        mem32[0x90008] = 0x00000003
        r0 = 0x00000000
        r1 = 0x00090000
        r4 = 0x00000004

        LDR  r0, [r1, r4]     ; register

POST    r0 = 0x00000002
        r1 = 0x00090000
        r4 = 0x00000004
```

Endereçamento com Load/Store

Addressing modes.

Type	Syntax
Load/store register	[Rn, Rm]
Base register + offset	[Rn, #immediate]
Relative	[pc|sp, #immediate]

```
PRE     mem32[0x90000] = 0x00000001
        mem32[0x90004] = 0x00000002
        mem32[0x90008] = 0x00000003
        r0 = 0x00000000
        r1 = 0x00090000
        r4 = 0x00000004

        LDR    r0, [r1, #0x4]    ; immediate

POST    r0 = 0x00000002
```

Load/Store Multiple Register

- Apenas suporta endereçamento com incremento após (IA)
  - Sempre atualiza a base (Rn)

```
Syntax : <LDM|STM>IA Rn!, {low Register list}
```

LDMIA	load multiple registers	{Rd}*N <- mem32[Rn + 4*N], Rn = Rn + 4*N
STMIA	save multiple registers	{Rd}*N -> mem32[Rn + 4*N], Rn = Rn + 4*N

Load/Store Multiple Register

- Exemplo:

```
PRE     r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003
        r4 = 0x9000

        STMIA   r4!,{r1,r2,r3}

POST    mem32[0x9000] = 0x00000001
        mem32[0x9004] = 0x00000002
        mem32[0x9008] = 0x00000003
        r4 = 0x900c
```

Operações com a pilha

- Utiliza-se PUSH e POP
  - R13 é sempre o SP
  - Utiliza apenas os registradores R0-R7, lr (PUSH) e pc (POP)
  - Operação é sempre FD (full descending)
  - POP$\rightarrow$ Retirados da pilha em ordem crescente
  - PUSH $\rightarrow$Colocados na pilha em ordem decrescente

Operações com a pilha

```
Syntax: POP {low_register_list{, pc}}
        PUSH {low_register_list{, lr}}
```

POP	pop registers from the stacks	Rd*N <- mem32[sp + 4*N], sp = sp + 4*N
PUSH	push registers on to the stack	Rd*N -> mem32[sp + 4*N], sp = sp − 4*N

Operações com a pilha

- Exemplo:

```
        ; Call subroutine
        BL      ThumbRoutine
        ; continue

ThumbRoutine
        PUSH    {r1, lr}        ; enter subroutine
        MOV     r0, #2
        POP     {r1, pc}        ; return from subroutine
```

Software Interrupt

- Equivalente ao modo ARM, mas o número da interrupção é limitado ao intervalo de 0 a 255.

```
Syntax: SWI immediate
```

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts) *cpsr T* = 0 (ARM state)

Software Interrupt

- Se houver interrupção ou execeção, o processador volta para o modo ARM

```
PRE     cpsr = nzcVqifT_USER
        pc = 0x00008000
        lr = 0x003fffff          ; lr = r14
        r0 = 0x12

    0x00008000    SWI    0x45
```

```
POST    cpsr = nzcVqIft_SVC
        spsr = nzcVqifT_USER
        pc = 0x00000008
        lr = 0x00008002
        r0 = 0x12
```

QXD0133 - Arquitetura e Organização de Computadores II

Universidade Federal do Ceará - Campus Quixadá

Thiago Werlley
thiagowerlley@ufc.br

18 de outubro de 2025

Capítulo 4