Arquiteturas

Arquiteturas de Computadores 1/3

Glauber De Bona

PCS - Departamento de Engenharia de Computação e Sistemas Digitais
Escola Politécnica da Universidade de São Paulo

Setembro, 2020

Agenda

Projeto de Sistemas Digitais

Nas disciplinas de SD, vimos diferentes técnicas de projeto:

- Circuitos combinatórios: álgebra de chaveamento, tabela verdade, mapas de Karnaugh
- Circuitos sequenciais síncronos: máquinas de estado finito, circuito de próximo estado e de saída
- Sistemas que implementam algoritmos (**não-programáveis**): diagrama ASM de alto nível, fluxo dados e unidade de controle

Como projetar um processador (**programável**)?
Partimos dos comandos possíveis para a sua programação!

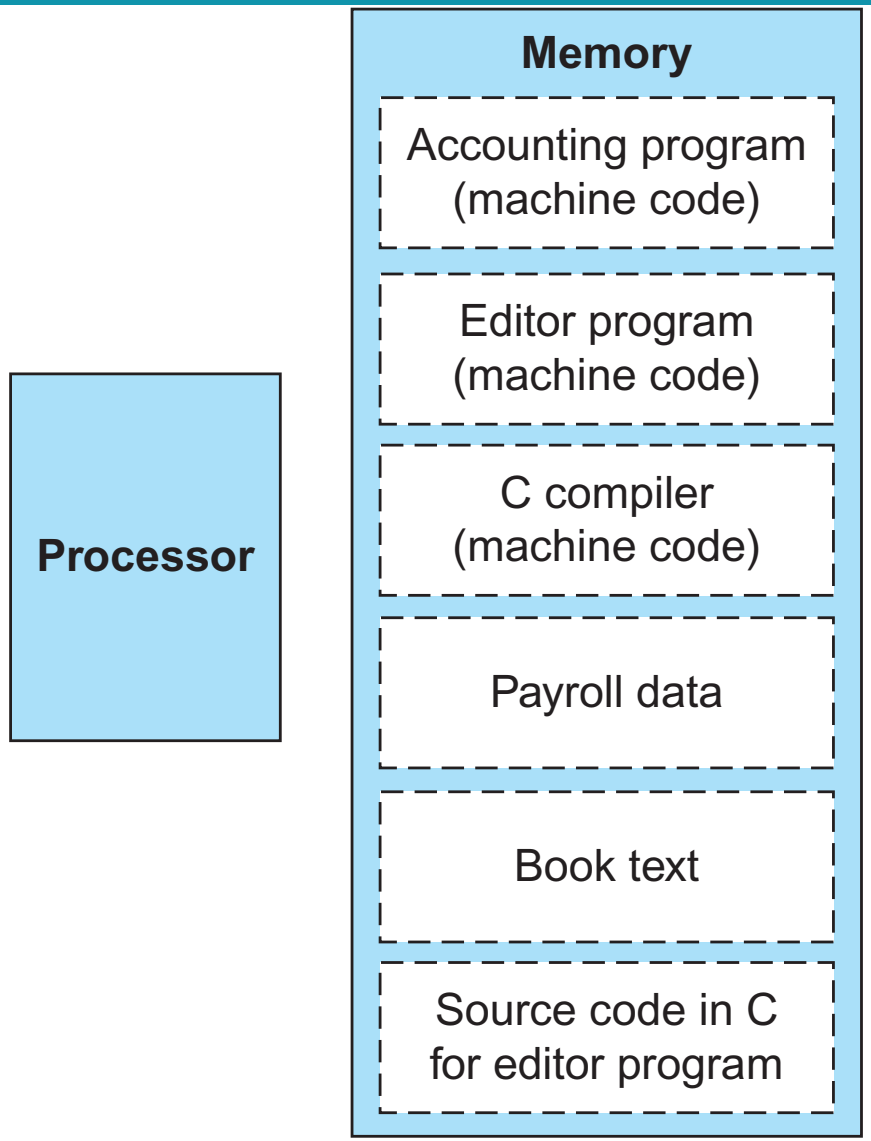
Arquitetura do Conjunto de Instruções

- Cada comando que um computador executa é chamado de **instrução**.
- O vocabulário do computador é o **conjunto de instruções** que ele executa.
- A **Arquitetura do Conjunto de Instruções** define a semântica das instruções.
  - engloba definições do tipo de dados, registradores e memória
  - é a interface entre hardware e software
  - pode ser implementada por diferentes processadores
- Uma vez que você aprenda uma **arquitetura de computador** específica, será fácil aprender outras.

Programa Armazenado

- Na memória, tudo são bits.
- Os bits podem ser considerados:
  - Dados: números, caracteres, *pixels*, etc.
  - Instruções: comandos para o processador.
- Em conjunto, os dados e instruções formam um **programa armazenado** na memória.
- Isto permite que um programa seja modificado/copiado/enviado como qualquer dado no computador.

Programa Armazenado



LEGv8: Assembly e Linguagem de Máquina

- As instruções representadas bits formam a **linguagem de máquina** de um processador. Ex: 1011001010001100
- Usaremos uma representação simbólica, mnemônica, chamada de **assembly**. Ex: ADD X, Y
- Nesta disciplina, trabalharemos com um subconjunto de instruções do ARMv8, chamado LEGv8:
  - SD2: conjunto reduzido do LEGv8;
  - OAC1: conjunto expandido e *pipelining*;
  - OAC2: conjunto completo e arquiteturas avançadas.

Instrução

Considere o seguinte código:

```
1  ADD a, b, c
```

O que ele faz? Soma b com c e coloca o resultado em a.

Operandos

Considere o seguinte código:

```
1   ADD  a ,  b ,  c
```

Mas o que são e onde estão a, b e c? São variáveis de um programa e estão em registradores.

Operandos LEGv8

Instruções do LEGv8 têm dois tipos de operandos:

- Registradores (32 x 64 bits);
- Memória de dados (qualquer tamanho de palavra).

Name	Example	Comments
32 registers	X0–X30, XZR	Fast locations for data. In LEGv8, data must be in registers to perform arithmetic, register XZR always equals 0.
2^{62} memory words	Memory[0], Memory[4], . . . , Memory[4,611,686,018,427,387, 904]	Accessed only by data transfer instructions. LEGv8 uses byte addresses, so sequential doubleword addresses differ by 8. Memory holds data structures, arrays, and spilled registers.

Instruções LEGv8

As instruções que fazem parte do **ISA** (*Instruction Set Architecture*) do LEGv8 são divididas em:

- Aritméticas (ADD e SUB);
- Transferência de dados (LD, ST e MOV);
- Lógicas (AND, OR, xor (EOR), shift (LSL, LSR));
- Salto condicional (CB, B.n);
- Salto incondicional (B).

Instruções LEGv8

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	ADD X1, X2, X3	X1 = X2 + X3	Three register operands
	subtract	SUB X1, X2, X3	X1 = X2 − X3	Three register operands
	add immediate	ADDI X1, X2, 20	X1 = X2 + 20	Used to add constants
	subtract immediate	SUBI X1, X2, 20	X1 = X2 − 20	Used to subtract constants
	add and set flags	ADDS X1, X2, X3	X1 = X2 + X3	Add, set condition codes
	subtract and set flags	SUBS X1, X2, X3	X1 = X2 − X3	Subtract, set condition codes
	add immediate and set flags	ADDIS X1, X2, 20	X1 = X2 + 20	Add constant, set condition codes
	subtract immediate and set flags	SUBIS X1, X2, 20	X1 = X2 − 20	Subtract constant, set condition codes
Data transfer	load register	LDUR X1, [X2,40]	X1 = Memory[X2 + 40]	Doubleword from memory to register
	store register	STUR X1, [X2,40]	Memory[X2 + 40] = X1	Doubleword from register to memory
	load signed word	LDURSW X1,[X2,40]	X1 = Memory[X2 + 40]	Word from memory to register
	store word	STURW X1, [X2,40]	Memory[X2 + 40] = X1	Word from register to memory
	load half	LDURH X1, [X2,40]	X1 = Memory[X2 + 40]	Halfword memory to register
	store half	STURH X1, [X2,40]	Memory[X2 + 40] = X1	Halfword register to memory
	load byte	LDURB X1, [X2,40]	X1 = Memory[X2 + 40]	Byte from memory to register
	store byte	STURB X1, [X2,40]	Memory[X2 + 40] = X1	Byte from register to memory
	load exclusive register	LDXR X1, [X2,0]	X1 = Memory[X2]	Load; 1st half of atomic swap
	store exclusive register	STXR X1, X3 [X2]	Memory[X2]=X1;X3=0 or 1	Store; 2nd half of atomic swap
	move wide with zero	MOVZ X1,20, LSL 0	X1 = 20 or 20 * 2^{16} or 20 * 2^{32} or 20 * 2^{48}	Loads 16-bit constant, rest zeros
	move wide with keep	MOVK X1,20, LSL 0	X1 = 20 or 20 * 2^{16} or 20 * 2^{32} or 20 * 2^{48}	Loads 16-bit constant, rest unchanged

Instruções LEGv8

Logical	and	`AND X1, X2, X3`	`X1 = X2 & X3`	Three reg. operands; bit-by-bit AND
	inclusive or	`ORR X1, X2, X3`	`X1 = X2 | X3`	Three reg. operands; bit-by-bit OR
	exclusive or	`EOR X1, X2, X3`	`X1 = X2 ^ X3`	Three reg. operands; bit-by-bit XOR
	and immediate	`ANDI X1, X2, 20`	`X1 = X2 & 20`	Bit-by-bit AND reg. with constant
	inclusive or immediate	`ORRI X1, X2, 20`	`X1 = X2 | 20`	Bit-by-bit OR reg. with constant
	exclusive or immediate	`EORI X1, X2, 20`	`X1 = X2 ^ 20`	Bit-by-bit XOR reg. with constant
	logical shift left	`LSL X1, X2, 10`	`X1 = X2 << 10`	Shift left by constant
	logical shift right	`LSR X1, X2, 10`	`X1 = X2 >> 10`	Shift right by constant
Conditional branch	compare and branch on equal 0	`CBZ X1, 25`	`if (X1 == 0) go to PC + 100`	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	`CBNZ X1, 25`	`if (X1 != 0) go to PC + 100`	Not equal 0 test; PC-relative branch
	branch conditionally	`B.cond 25`	`if (condition true) go to PC + 100`	Test condition codes; if true, branch
Unconditional branch	branch	`B 2500`	`go to PC + 10000`	Branch to target address; PC-relative
	branch to register	`BR X30`	`go to X30`	For switch, procedure return
	branch with link	`BL 2500`	`X30 = PC + 4; PC + 10000`	For procedure call PC-relative

Exemplo 1

Considere o seguinte trecho de código:

```
1  a = b + c;
2  d = a - e;
3  x = y - 2;
4  z = w + 4;
```

O código (assembly) gerado para o LEGv8 será:

```
1  ADD a, b, c
2  SUB d, a, e
3  SUBI x, y, 2
4  ADDI z, w, 4
```

Exemplo 2

Considere o seguinte trecho de código:

```
1   f = (g + h) - (i + 3);
```

Qual o código assembly para o LEGv8?

Exemplo 2

Considere o seguinte trecho de código:

```
1  f = (g + h) - (i + 3);
```

O código assembly gerado para o LEGv8 será:

```
1  ADD t0,g,h   // t0 = g + h
2  ADDI t1,i,3   // t1 = i + 3
3  SUB f,t0,t1 // f = t0 - t1 = (g+h) - (i+3)
```

Operandos

Mas onde estão as variáveis?

- Toda variável é mapeada em um registrador;
- Cada registrador armazenada uma *doubleword*.
  - *doubleword* (64 bits = 8 bytes)
  - *word* (32 bits = 4 bytes)
- O valor de uma variável é trazido da memória para um registrador antes da operação e fica no registrador até não ser mais necessário.

Quantas variáveis podem "existir" ao mesmo tempo em um código compilado para LEGv8?

Exemplo 3

Em uma instrução, X0,X1,...,X30,XZR representam os registragores; #15 e # − 87 são constantes.

Considere o seguinte trecho de código (revisitado):

```
1   f = (g + h) - (i + 3);
```

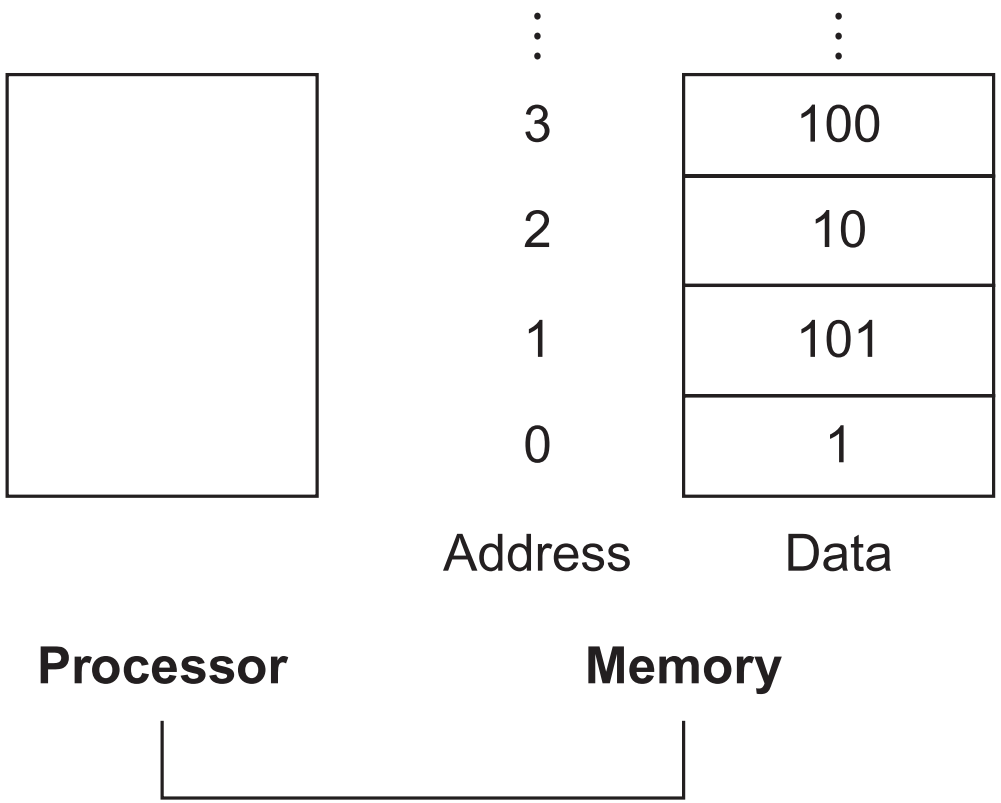
O código assembly gerado para o LEGv8 será:

```
1   ADD X9,X20,X21  // X9=g+h=X20+X21
2   ADDI X10,X22,#3  // X10=i+3=X22+3
3   SUB X19,X9,X10  // f=X9-X10=(g+h)-(i+3)
```

Mas de onde vem os valores dos registradores?

Trazendo Palavras

Há instruções específicas para trazer dados da memória para os registradores.
Exemplo de uma memória:



Exemplo 4

Considere que `A` é um vetor de *doublewords* cujo endereço base está em `X22`:

```
1   g = h + A[8]
```

O código assembly gerado para o LEGv8 será:

```
1   LDUR  X9,[X22,#8]  //  X9=mem[X22+64]
2   ADD   X20,X21,X9   //  g=h+X9
```

- `X22` é chamado de registrador base (*base register*);
- 8 é chamado de *offset*;
- `X9` recebe `A[8]`
- `LDUR` significa *LoaD Unscaled immediate to Register*
- **Qual a palavra carregada pelo load?**

Exemplo 5

Considere o mesmo vetor do exemplo anterior:

```
1   A[12] = h + A[8];
```

O código assembly gerado para o LEGv8 será:

```
1   LDUR X9, [X22,#64] // Reg X9 recebe A[8]
2   ADD  X9,X21,X9     // Reg X9 recebe h + A[8]
3   STUR X9, [X22,#96] // Grava h + A[8] em A[12]
```

- As palavras são *doublewords*, portanto cada uma tem 8 bytes.
- Ao acessar a palavra no endereço `X22+64`, estaremos acessando a oitava *doubleword* do vetor.

Limitações

Mas só podemos escrever programas com até 31 variáveis no mesmo escopo?

- Podemos salvar os valores dos registradores na memória (*spilling*);
- Podemos trazer valores salvos de volta da memória.
- É papel do compilador alocar corretamente as variáveis para os registradores para minimizar o *spilling*.
- Alguns registradores possuem propósito especial e normalmente não são alocados para variáveis. Na prática, 8 dos 32 registradores são livres para uso geral.

Registradores LEGv8

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
X0 – X7	0-7	Arguments / Results	No
X8	8	Indirect result location register	No
X9 – X15	9-15	Temporaries	No
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	No
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	No
X18	18	Platform register for platform independent code; otherwise a temporary register	No
X19-X27	19-27	Saved	Yes
X28 (SP)	28	Stack Pointer	Yes
X29 (FP)	29	Frame Pointer	Yes
X30 (LR)	30	Return Address	Yes
XZR	31	The Constant Value 0	N.A.

Exercício 1

Considere o seguinte trecho de código e a memória ao lado. Assuma que a doubleword no endereço 0 é $00\ldots00FF$.

```
1    ADD   X22,XZR,XZR
2    LDUR  X9,[X22,#0]
3    LDUR  X10,[X22,#16]
4    ADD   X9,X9,X10
5    SUB   X10,X9,X10
6    LDUR  X11,[X22,#8]
7    ADD   X9,X11,X9
```

7	FF	F	00	17	01	1F	01
6	00	E	01	16	00	1E	FF
5	00	D	00	15	00	1D	00
4	00	C	00	14	00	1C	00
3	00	B	00	13	00	1B	00
2	00	A	00	12	00	1A	00
1	00	9	00	11	00	19	00
0	00	8	00	10	00	18	00

Qual o conteúdo de `X9` após a execução?

Exercício 1 - Solução

Considere o seguinte trecho de código e a memória ao lado. Assuma que a doubleword no endereço 0 é $00\ldots00FF$.

```
1   ADD   X22,XZR,XZR
2   LDUR  X9,[X22,#0]
3   LDUR  X10,[X22,#16]
4   ADD   X9,X9,X10
5   SUB   X10,X9,X10
6   LDUR  X11,[X22,#8]
7   ADD   X9,X11,X9
```

7	FF	F	00	17	01	1F	01
6	00	E	01	16	00	1E	FF
5	00	D	00	15	00	1D	00
4	00	C	00	14	00	1C	00
3	00	B	00	13	00	1B	00
2	00	A	00	12	00	1A	00
1	00	9	00	11	00	19	00
0	00	8	00	10	00	18	00

- Primeiro zeramos o X22, então X9=FFh=255
- 16=10h, então X10=1 e X9=X9+X10=256
- X11=100h=256, e X9=X11+X9=512

Exercício 2

Considere o seguinte trecho de código e a memória ao lado.
Assuma que a doubleword no endereço 0 é $00\ldots00FF$.

```
1   SUB   X22,X22,X22
2   ADDI  X22,X22,#16
3   LDUR  X9,[X22,#0]
4   SUB   X10,X22,X9
5   STUR  X10,[X22,#-8]
6   LDUR  X9,  [XZR,#8]
```

7	FF	F	00	17	01	1F	01
6	00	E	01	16	00	1E	FF
5	00	D	00	15	00	1D	00
4	00	C	00	14	00	1C	00
3	00	B	00	13	00	1B	00
2	00	A	00	12	00	1A	00
1	00	9	00	11	00	19	00
0	00	8	00	10	00	18	00

Qual o conteúdo de `X9` após a execução?

Exercício 2 - Solução

Considere o seguinte trecho de código e a memória ao lado. Assuma que a doubleword no endereço 0 é $00\ldots00FF$.

```
1   SUB   X22,X22,X22
2   ADDI  X22,X22,#16
3   LDUR  X9,[X22,#0]
4   SUB   X10,X22,X9
5   STUR  X10,[X22,#-8]
6   LDUR  X9,  [XZR,#8]
```

7	FF	F	00	17	01	1F	01
6	00	E	01	16	00	1E	FF
5	00	D	00	15	00	1D	00
4	00	C	00	14	00	1C	00
3	00	B	00	13	00	1B	00
2	00	A	00	12	00	1A	00
1	00	9	00	11	00	19	00
0	00	8	00	10	00	18	00

- Primeiro zeramos o X22, então X22=0+16=16
- 16=10h, então X9=1 e X10=16-1=15
- Como 16-8=8, armazenamos X10 em 8, de onde carregamos X9=15.

Exercício 3

Gere o código de máquina correspondente ao código C abaixo. Se necessário, use o arquivo GreenCard.pdf disponível no e-disciplinas. O `f` deve ser guardado na memória depois (você escolhe o endereço).

```
1    f = a-b+v1[0]+v2[3]-4
```

Exercício 3 - Solução

Gere o código assembly correspondente ao código C abaixo. Os elementos dos vetores têm 64 bits. O `f` deve ser guardado na memória depois (você escolhe o endereço).

```
1    f = a-b+v1[0]+v2[3]-4;
```

```
SUB X9,X10,X11          \\ f:X9,a:X10,b:X11
LDUR X12,[X22,#0]       \\ v1:X22, X12:temp
ADD X9,X9,X12
LDUR X12,[X23,#24]      \\ v2:X23
ADD X9,X9,X12
SUBI X9,X9,4
STUR X9,[XZR,#48]
```

Obrigado!


Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais
PCS

Referências


D. Patterson and J. Hennessy.
Computer Organization and Design ARM Edition: The Hardware Software Interface.
The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2016.