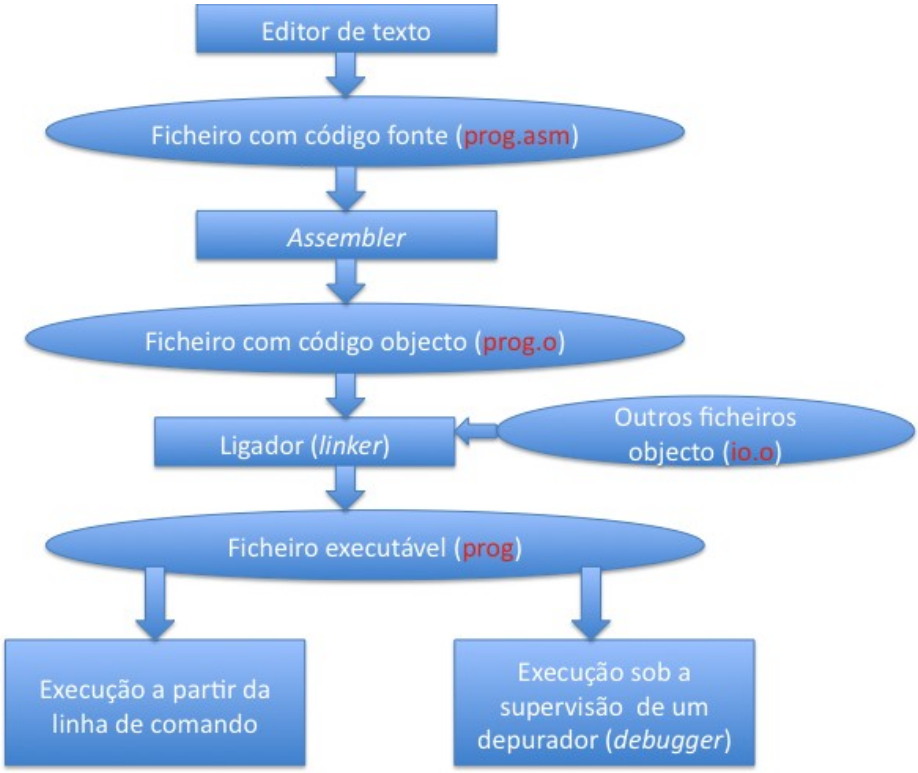
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA
Departamento de Informática

Introdução aos Sistemas e Redes de Computadores 2009/10

Assuntos: O processador PENTIUM como exemplo de um CPU real; contacto com um sistema de desenvolvimento que inclui o "assembler" *nasm*, um ligador e um depurador (debugger) *gdb* (e a sua interface gráfica *ddd)* ; escrita de pequenos programas em "assembly" do Pentium

A. Desenvolvimento de um programa em "assembly"

O ciclo de desenvolvimento de programas em "assembly" está resumido na figura seguinte:



As ferramentas que aparecem na figura são:

- **Editor de texto**: para escrever o código fonte de um programa em *assembly* pode ser usado qualquer editor de texto que permita salvar o conteúdo editado como uma sequência de caracteres (normalmente códigos ASCII); não podem ser usados qualquer tipo de atributos de formatação

  - ***Assembler***: é um tradutor que permite converter um programa escrito em linguagem *assembly* característica de um dado CPU nos respectivos códigos máquina. A linguagem *assembly* tem um nível de abstracção muito próximo do da linguagem máquina: de uma forma aproximada cada linha do programa escrito em *assembly* corresponde a uma instrução máquina. A escrita em *assembly* permite:

    - Especificar as instruções por **mnemónicas** em vez de sequências de bits ou códigos numéricos;
    - Especificar os operandos das instruções de forma simbólica:

        - Se são posições de memória é possível designá-las por nomes (etiquetas) e não por números (endereços); isto corresponde aproximadamente, nalguns aspectos, à noção de **variável** numa linguagem de alto nível
        - Se são constantes é possível designá-las por nomes em vez de explicitamente indicar o valor; numa linguagem de alto nível isto corresponde ao que se chama uma **constante**
    - Usar **pseudo-instruções**: um programa em *assembly* contém linhas que não correspondem a instruções máquina (daí o qualificativo pseudo). Essas linhas contêm indicações dadas pelo programador ao assembler e que condicionam a operação desse programa (ver exemplos concretos à frente)

- **Ligador (linker ou link editor)**: é um programa de sistema que permite desenvolver programas constituídos por módulos separados. Para que esses módulos possam ser desenvolvidos separadamente é preciso que um módulo possa fazer referência a símbolos (código ou dados) definidos noutro módulo. Para esse efeito, os símbolos de um programa podem ter um dos seguintes qualificativos:
    - Público: se esse símbolo puder ser referenciado de um outro módulo.
    - Externo: este qualificativo informa o *assembler* que não deve procurar a definição do símbolo no módulo que está a processar. A definição do símbolo ficará para mais tarde.

  Quem faz a resolução dos símbolos externos é o **ligador**. Este programa vai associar os símbolos externos de um módulo a símbolos públicos de outros módulos. Esta associação de nomes a valores (endereços) é feita pelo ligador permitindo a concatenação dos diferentes ficheiros **objecto** (módulos) num único ficheiro **executável**. Este será o programa final completo caso todas as resoluções dos nomes externos tenham sucesso.

- **Depurador (debugger)**: é um programa que controla a execução de outro, permitindo a sua execução por etapas e a obtenção de informação sobre o seu estado. Um *depurador simbólico* permite parar a execução do programa e observar o valor corrente de variáveis e o efeito da execução de cada instrução; no caso de um programa em *assembly* permite ver o conteúdo dos registos do processador e de posições de memória.

Na secção C deste documento introduzem-se resumidamente os principais aspectos de um assembler (nasm), ligador (ld) e de dois depuradores (gdb e ddd). A descrição feita assume que as ferramentas executam em ambiente Linux.

B. Algumas aspectos relevantes do Pentium

Neste ponto, referem-se alguns aspectos da arquitectura do Pentium e das instruções máquina. Este resumo segue de perto a aula teórica de 15 de Dezembro de 2009 *[AULA_15Dez]*. Chama-se a atenção que não se trata de uma descrição exaustiva, mas apenas da enumeração de alguns aspectos importantes e que são suficientes para a resolução dos exercícios propostos. Utiliza-se a sintaxe do "assembler" **nasm** descrito na secção C.

1. Alguns do registos do processador (todos de 32 bits)

EAX, EBX, ECX, EDX

- Registos de uso geral, podem conter dados ou endereços de memória.

- Algumas instruções máquina envolvem implicitamente alguns destes registos (por exemplo multiplicação, divisão e LOOP)

- Para operações a 16 bits estes registos são designados por AX, BX, CX e DX; nesta situação são considerados os 16 bits menos significativos dos registos EAX, EBX, ECX, EDX, respectivamente.

- Para operações a 8 bits cada um destes quatro registos pode ainda ser subdivido: os bits 0 a 7 do registo EAX constituem o registo AL e os bits 15 a 8 o registo AH; de forma análoga são definidos os registos BH e BL (subdivisão de BX), CH e CL (subdivisão de CX) e DH e DL (subdivisão de DX)

EIP: apontador para a próxima instrução a executar

ESP : "stack pointer", aponta para o topo da pilha em memória

Códigos de condição (1 bit cada)

ZF (Zero Flag) : quando está a VERDADE (1) indica se a última operação aritmética ou lógica deu resultado 0; se está a falso (0) significa que a essa última operação não deu resultado 0.

PF(Parity Flg) : Se está a 1 indica que a última operação aritmética ou lógica deu resultado par; se está a 0 indica que o resultado foi ímpar.

CF(Carry Flag) : Se está a 1 indica que a última operação aritmética deu transporte (*carry*); se está a 0 indica que não houve *carry*; numa operação a 32 bits indique que a operação elementar sobre o bit 31 deu transporte.

SF (Sign Flag) : SF=1 indica se o resultado da última operação aritmética e lógica tinha o bit mais significativo a 1 (ie em aritmética de complemento a 2 é negativo); se SF=0 significa que o bit de sinal é 0.

OF(Overflow Flag) : Se OF=1 significa que a última operação aritmética efectuada deu um resultado que excede a capacidade de representação da máquina; por exemplo se se somaram dois números positivos representados em complemento para 2 e o resultado deu negativo, isso significa que o resultado era um número positivo que não podia ser representado no número de bits disponível.

2. Instruções de movimentação de dados

Nas descrições seguintes são usadas as convenções e as mnemónicas do *assembler* nasm já referido e cuja operação é descrita na secção C.

As instruções de movimento de dados (load e store) têm todas a mnemónica *mov* e obedecem ao seguinte padrão

mov destino, fonte

O valor designado pelo operando *fonte* é copiado para o registo ou posição de memória que é designado por *destino*. É o equivalente a uma instrução de afectação numa linguagem de alto nível

DESTINO = Conteúdo guardado na FONTE

- Fonte pode designar:
  - uma constante
  - o conteúdo de um registo do CPU
  - o conteúdo de uma posição de memória
- Destino pode ser:
  - um registo do CPU
  - uma posição de memória

A combinação em que a fonte e o destino são posições de memória não é possível

As constantes podem ser escritas em decimal (por omissão), hexadecimal (sufixo H ou h ou prefixos 0x ou $), octal (sufixo O,Q,o ou o) ou binário (sufixo b ou B). Exemplos:

- 100 (decimal)
- 0a2h, $0a2, 0xa2 (hexadecimal)
- 777q, 777o (octal)
- 10010011b, 1001_0011B (binário)

Os registos são designados pelos nomes acima referidos (EAX, ...). É indiferente ser em maiúsculas ou minúsculas.

As posições de memória são designadas de várias formas (**modos de endereçamento**). Quando a *fonte* ou o *destino* é uma posição de memória, o operando é o respectivo endereço ou uma expressão cuja avaliação fornece o endereço efectivo de memória onde se encontre o operando. Há várias formas de calcular esse endereço; indicam-se a seguir algumas. A explicação baseia-se no seguinte fragmento de programa que usa a sintaxe do assembler *nasm*:

```
letter    db  ´A´          ; reserva de uma zona de memória com um byte preenchida
                           ; com o código ASCII do 'A'  ou seja 0x41; o nome simbólico é letter
table1    resw  20         ; reserva de 20 palavras com 16 bits não inicializado

name1  db  'isrc',0        ; 5 bytes inicializados com os códigos ASCII correpondentes e
                           ; terminadas por 0
```

Directo: o endereço é directamente colocado na instrução, podendo ser a etiqueta da posição de memória, entre parêntesis rectos.

```
mov  al, [letter]      ; al recebe o conteúdo da posição letter, ie 'A'
mov  [letter], 'C'     ; na posição letter é guardado 'C'
```

Indirecto por registo : o endereço está guardado num registo do CPU

```
mov  ebx, table1       ; ebx recebe o endereço da posição table1
mov  [ebx], 100        ; na posição table1 é guardado 100
```

Baseado: o endereço efectivo é calculado somando o valor de uma constante com o que está contido um registo

```
mov  ecx, name1        ; ecx recebe o endereço da posição name1
mov  al, [ecx+2]       ; al recebe 'r'
mov  [ecx+3], 'C'
```

É muito importante perceber a diferença entre

`mov al, [letter]` em que se faz o acesso ao conteúdo da posição cuja etiqueta é *letter*, e

`mov ebx, letter` em que se carrega no registo ebx um valor com 32 bits que é o endereço da posição *letter,*

Instruções aritméticas e lógicas com inteiros

As instruções aritméticas e lógicas do Pentium podem dividir-se em três grupos

- instruções com dois operandos: *OP2 operando1, operando2*. Nestas operações é que operando1 = operando1 OP2 operando2, isto é o 1º operando também funciona como destino do resultado
- instruções com um operando: OP1 operando. operando = OP1 operando

- multiplicação e divisão de inteiros: As instruções de multiplicação e divisão não obedecem a este esquema. Só têm um operando explícito. Implicitamente o registo EAX (e nalguns casos o EDX) intervém sempre. Consulte as referência *[Intel_Manual]* ou *[Art_of_Assembly]* para detalhes.

As instruções aritméticas e lógicas afectam o valor das *flags* de acordo com o resultado e servem para tomar decisões quando se executam saltos condicionais (ver à frente)

Principais operações com um operando

INC op : op = op +1; a *flag* CF não é afectada

DEC op: op = op -1; a *flag* CF não é afectada

NOT op : o operando recebe o complemento para 1 de cada um dos seus bits

NEG op: o operando recebe o seu complemento para 2

Principais operações com dois operandos

ADD op1, op2 : op1 = op1 + op2

SUB op1, op2: op1 = op1 - op2

CMP op1, op2: é feita a operação op1 - op2 mas o resultado perde-se; são afectadas as *flags*

AND op1, op2 : op1 = op1 AND op2; nesta instrução e seguintes a operação é feita bit a bit

OR op1, op2 : op1 = op1 OR op2

XOR op1, op2 : op1 = op1 XOR op2

TEST op1, op2: é feita a operação op1 AND op2 mas o resultado perde-se; são afectadas as *flags*

Rotações e deslocamentos
Consulte a referência *[Intel_Manual]*.

Saltos incondicionais e condicionais

Saltos incondicionais
A mnemónica desta instrução é JMP. O valor do EIP (Instruction Pointer) é modificado, recebendo o endereço argumento da instrução. Tal endereço pode ser indicado usando uma etiqueta. Exemplo:

```
L1:     ...
        ...
        jmp   L1
```

Saltos condicionais
Estas instruções têm o formato
J<condição> etiqueta
e testam o valor de um condição lógica em que intervém o valar de uma ou mais *flags*. Se a condição for verdadeira o IP é modificado recebendo o endereço que corresponde à etiqueta que é o argumento da instrução.
No exemplo abaixo, as instruções entre L2 e jz L2 são executadas até ecx se tornar igual a 100.

```
        xor         ecx, ecx    ; forma compacta de colocar ecx a 0
L2:     ...
        inc         ecx
        cmp     ecx, 100
        jnz         L2
```

Há um grande número de saltos condicionais (não se esqueça que se assume que houve uma instrução aritmética e lógica prévia que afectou as *flags*)

Teste directo de flags

- **jc** jump if carry flag (ie saltar se CF igual a 1)
- **jnc** jump if not carry flag (ie slatar se CF igual a 0)
- **jz** jump if zero flag (ie saltar se ZF igual a 1)
- **jnz** jump if not zero flag (ie saltar se ZF igual a 0)

Teste de condições baseadas em uma ou mais flags de acordo com a aritmética com sinal (a seguir a um **cmp op1,op2**)

- **jl** saltar se for verdade que op1 < op2
- **jle** saltar se for verdade que op1 <= op2
- **jg** saltar se for verdade que op1 > op2
- **jge** saltar se for verdade que op1 >= op2
- **je** saltar se for verdade que op1 = op2

Refira-se ainda que há uma instrução máquina que permite implementar facilmente ciclos. É a instrução LOOP etiqueta. Essa instrução decrementa implicitamente o registo ecx e salta para a etiqueta se ECX for maior do que 0. Veja o exemplo seguinte, em que ecx funciona como contador do ciclo que é cumprido 100 vezes.

```
        mov     ecx, 100
L3:     ...
        ...
        loop    L3
```

Pilha ou stack

O Pentium tem uma pilha que segue a disciplina de inserção "Last In First Out". A pilha é gerida usando um registo do CPU dedicado chamado ESP. O ESP aponta para a última posição ocupada na pilha. Vamos supor que são sempre empilhados 4 bytes (32 bits). As principais operações são ***PUSH e POP*** :

Push valor

O valor pode ser uma constante, o conteúdo de um registo ou o conteúdo de uma posição de memória.

SP = SP - 4
mem[SP] = valor $_{7\text{-}0}$; bits 7 a 0 do valor a empilhar

mem[SP+1] = valor $_{15\text{-}8}$

mem[SP+2] = valor $_{23\text{-}16}$

mem[SP+3] = valor $_{31\text{-}24}$

Pop destino
O destino pode ser .um registo ou uma posição de memória
destino $_{7\text{-}0}$ = mem[SP] ; bits 7 a 0 do destino

destino $_{15\text{-}8}$ = mem[SP+1]

destino $_{23\text{-}16}$ = mem[SP+2]

destino $_{31\text{-}24}$ = mem[SP+3]
SP = SP + 4

Subrotinas

Chamada de subrotina: instrução CALL etiqueta
Esta instrução faz push do EIP e escreve o valor associado à etiqueta no EIP

Retorno de subrotina; instrução RET
Esta instrução é o equivalente a um pop EIP, isto é, espera-se que o valor que foi empilhado pela instrução CALL seja desempilhado do stack e carregado no EIP. Assim a execução continua na instrução máquina a seguir à instrução CALL que invocou a subrotina corrente

Passagem de parâmetros
Os parâmetros podem ser passados por registos ou empilhados no stack usando a instrução PUSH. Se os parâmetros foram empilhados, pode-se usar endereçamento baseado (usando o ESP como registo base) para fazer acesso ao seu valor . O último parâmetro empilhado tem o endereço [esp+4], o penúltimo [esp+8], etc.

Retorno de resultados
Se há só um valor a retornar é usado o registo eax para esse efeito

O exemplo seguinte ilustra o que foi dito:

```
        push   2
        push   [val]
        call   my_add              ; my_add( val, 2)
                                          ; eax tem o valor retornado
        ……
my_add:     mov    eax, [esp+4]
        mov    ebx,[esp+8]
        add    eax, ebx
        ret
```

NOP
Vale a pena referir a existência das duas seguintes instruções
NOP: esta instrução não tem qualquer efeito
HLT: é suspenso o processamento de instruções

Para mais detalhes sobre o conjunto de instruções do Pentium veja as referencias *[Intel_Manual]* **e** *[Art_of_Assembly]* **da secção F**.

C. O "assembler" NASM

O NASM é um *assembler* de domínio público que funciona em Windows, Linux e MAC OSX. Produz ficheiro objecto em variados formatos e tem um sintaxe semelhante aos *assemblers* da Intel. A descrição seguinte supõe que o NASM está a ser usado em LINUX e foca-se nos aspectos necessários à resolução dos problemas propostos. O manual completa é a referência *[NASM_DOC]* da secção F.

A invocação do NASM é feita na linha do comando do Linux da seguinte forma:

```
nasm –f elf32  –g  prog.asm
```

Supõe-se que existe um ficheiro *prog.asm* que tem um programa em *assembly* do Pentium com as convenções usadas pelo NASM; se o ficheiro existir e contiver um programa sintacticamente correcto é criado um ficheiro *prog.o*. A opção *–f elf32* indica que os ficheiros objecto são criados no formato ELF e a opção *–g* pede a inclusão no ficheiro objecto de informação sobre as etiquetas usadas o que permite a depuração simbólica (ver à frente).

Informação sobre outras opções disponíveis podem ser conseguidas fazendo

```
man nasm

info nasm
```

Pseudo instruções do NASM

Como atrás foi referido, cada assembler define a sintaxe e a semântica como uma linguagem de programação. Como indicado na secção B, a linguagem simbólica definida permite linhas que correspondem directamente a instruções máquina e outras que são as chamadas **pseudo-instruções**.

Como as convenções usadas nas linhas que contêm instruções já foram abreviadamente descritas na secção B, falta falar nas pseudo-instruções. Também aqui não se faz uma descrição exaustiva mas refere-se apenas o que permite compreender os exemplos e resolver os exercícios propostos

A listagem seguinte apresenta um programa completo em assembly do Pentium. Os comentários descrevem algumas das pseudo-instruções.

```
global _start      ; definição de um símbolo  que é visível do exterior do
                   ; módulo. Este símbolo em particular (_start) define o
                   ; ponto de entrada do programa, isto é onde começa
                   ; a execução

SYS_EXIT       equ    1          ; definição de uma constante
LINUXCALL      equ    0x80       ; definição de outra constante

section .text      ; marca que indica o início do código do programa

_start:
    mov    eax, SYS_EXIT
    mov    ebx, 5
    int            LINUXCALL
```

Edite este programa e crie um ficheiro chamado *prog1.asm*. Para executar este programa terá de dar os comandos no "shell" do LINUX abaixo indicados. Note-se que é necessário o uso do ligador *ld,* embora o programa seja constituído por apenas um módulo.

```
nasm –f elf32 –g  prog1.asm
ld –o prog1 prog1.o
./prog1
```

Como pode observar o programa termina sem fazer nada. De facto, as três linhas de código correspondem à invocação da chamada ao sistema EXIT que termina o programa.
Para já não é possível explicar em detalhe o que se está a passar, uma vez que a explicação da instrução *int LINUXCALL* faz parte da matéria teórica da semana que começa em 4 de Janeiro de 2010. Uma explicação simplificada é essa instrução é uma invocação dos serviços do sistema; no caso do LINUX essa invocação é semelhante à chamada de uma subrotina em que os parâmetros são passados nos registos:

EAX: identificação da chamada ao sistema feita
  0x01 exit
  0x03 read
  0x04 write
  ...
EBX: 1º parâmetro
ECX: 2º parâmetro
EDX: 3º parâmetro

No exemplo é invocada a chamada ao sistema com o código 0x01 que é *exit()*. Essa chamada tem um parâmetro (passado em *ebx*) que é o valor que o programa que termina retorna ao programa que o lançou.
Nste caso, o programa que lançou o *prog1* foi o *shell*. Na *shell* pode-se consultar o valor retornado ao shell na variável de ambiente *$?*. Assim:

```
prompt> ./prog1
prompt> echo $?
5
prompt>
```

O exemplo seguinte envolve também o uso de chamadas ao sistema e de mais algumas pseudo-instruções do NASM.

```
global _start

        SYS_EXIT    equ    0x01
        SYS_WRITE equ    0x04
        LINUXCALL  equ    0x80

section .data  ;   marca que indica o início do zona de dados inicializados
        mesg: db      'Hello world!',0x0a
        MESGSZ  equ  13

section .text
_start:
        mov    eax, SYS_WRITE     ; write(1, mesg, 13)
        mov    ebx, 01
        mov    ecx, mesg
        mov    edx, MESGSZ
        int             LINUXCALL

        mov    eax, SYS_EXIT     ; exit(5)
        mov    ebx, 5
        int             LINUXCALL
```

Edite e execute este programa. Os passos a realizar são os mesmos do programa anterior.

D. Depuradores *gdb* e *ddd*

O programa gdb é um depurador simbólico. Se o ficheiro executável contém informação sobre os endereços de memória a que correspondem os símbolos do programa (variáveis e linhas de código) é possível seguir de forma interactiva a execução de um programa.

Para invocar o gdb é necessário fazer apenas

```
gdb prog
```

Seguidamente entra-se numa sessão interactiva, em que normalmente os dois primeiros comandos são:

break <etiqueta ou número de linha>

run

O 1º comando instala um *breakpoint*: quando a execução do programa chega a um breakpoint, suspende-se a execução e podem-se dar comandos que permitem ver o valor de variáveis, registos, etc. O comando *run* incia a execução do programa sob controlo do *gdb*.

A lista seguinte foi extraída da Internet e apresenta alguns dos principais comandos do gdb.

Basics:

- `run [<args>]` - run the program with (optional) arguments
- `c` (`continue`) - continue execution
- `bt` - print a stack trace
- `frame <n>` - switch to frame
- `info locals` - show local variables
- `p <foo>` - print the value of a variable
- `x /fmt addr` - show data under a given addr using a given format
- `info breaks` - display current breakpoints
- `break <line-no>` - set breakpoint at a line number in current file e.g. `break 24`
- `break *<addr>` - set a breakpoint at a given addr e.g. `break *foo + 24` (set a breakpoint 24 bytes after the beginning of function foo
- `delete <n>`, `disable <n>`, `enable <n>` - delete/disable/enable breakpoint
-

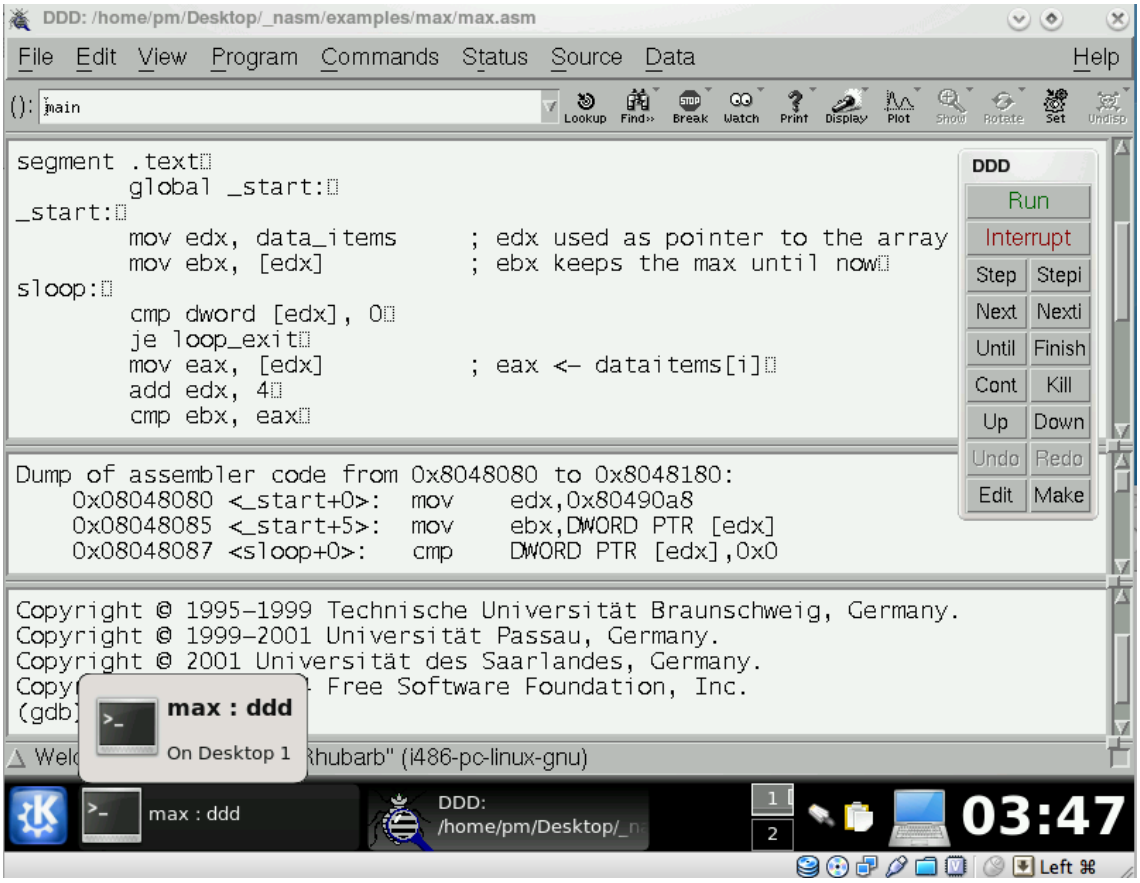
Debugging at assembly level:

- `display /i $eip` so that gdb prints the next assembly instruction
- `nexti` and `stepi` for stepping by one instruction
- `set disassembly-flavor intel` changes assembly syntax from awful AT&T to less awful intel
- `info registers` show content of registers

O programa ddd é uma interface gráfica do gdb. Invoca-se, fazendo no *shell*

```
ddd prog
```

A maior parte dos comandos acima referidos podem ser dados através de menus, botões, etc. simplificando a actividade do programador. A título de exemplo, apresenta-se um *screenshot* do ddd.



E. Uma biblioteca para ler e escrever no/do terminal

Como se viu nos programas até agora usados, há um conjunto de funcionalidades que vão sendo usadas repetidas vezes (chamada ao sistema exit, chamada para escrever uma cadeia no terminal, etc).

Considere-se o código seguinte:

```
global sysWrite, sysRead, sysExit0, putChar, getChar

;chamadas ao sistema de operação (Linux)
SYS_EXIT  equ 1
SYS_READ  equ 3
SYS_WRITE equ 4

;INTR de chamada ao sistema
LINUXCALL   equ 0x80

; sysWrite   - write bytes to standard output channel
; ecx <- address where write from
; edx <- number of bytes to write
sysWrite:
    push eax
    push ebx
    mov ebx, 1       ; write to standard output
    mov eax, SYS_WRITE
    int LINUXCALL
    pop ebx
    pop eax
    ret
```

```
; sysRead   - read bytes from standard input channel
; ecx <- address where to read
; edx <- max bytes to read
sysRead:
     push eax
     push ebx
     mov ebx, 0       ; read standard input
     mov eax, SYS_READ
     int LINUXCALL
     pop ebx
     pop eax
     ret


; sysExit0    - terminate program (return 0 to OS)
sysExit0:
     push eax
     push ebx
     mov ebx, 0
     mov eax, SYS_EXIT
     int LINUXCALL
     hlt        ; in case something goes wrong....

;=====================================================

; putChar    - write 1 char to standard output
; al <- char to write
putChar:
     push ecx
     push edx
     push eax         ; al has ASCII to write
     mov ecx, esp
     mov edx, 1       ; write just 1 char
     call sysWrite
     pop eax
     pop edx
     pop ecx
     ret


; getChar    - read 1 char from standard input
; al <- returns readed char (eax is changed)
getChar:
     push ecx
     push edx
     push eax         ; space where to read
     mov ecx, esp
     mov edx, 1       ; read just 1 char
     call sysRead
     pop eax                    ; put char in al
     pop edx
     pop ecx
     ret
```

Salve este código num ficheiro chamado *io.asm*. Como se pode ver, este código implementa um conjunto de operações sysWrite, sysRead, sysExit0, putChar, getChar que declara como públicas (directiva *global*). Essas funções podem ser chamadas do seu programa que fica agora muito mais simples.

```
extern sysExit0, sysWrite
global _start
LF  equ  0xA        ; LineFeed ascii code (new line)

section .data
    mesg:  db       'Hello world!', LF
    MESGSZ  equ  13

section .text
_start:
    mov ecx, mesg
    mov edx, MESGSZ
    call sysWrite

    call sysExit0
```

Note que o programa se tornou muito simples quando comparado com o que foi antes apresentado e tem funcionalidade semelhante. Note também a pseudo-instrução *extern* que indica ao assembler que os símbolos *sysExit0 e sysWrite* estão definidos noutro ficheiro.

Para obter o executável é preciso fazer:

```
nasm –f elf32 –g io.asm
nasm –f elf32 –g prog.asm
ld –o prog prog.o io.o
./prog
```

F. Referências

[AULA_15Dez] Pedro D. Medeiros, **Slides da aula teórica, 15 de Dezembro de 2009**, disponível em https://clip.unl.pt/objecto?oid=78641&oin=19-(2009.12.15)-ArquitecturaDeComputadores.ppt

[Intel_Manual] **Pentium Processor Family Developer´s Manual, Volume3: Architecture and Programming Manual**, capítulos 3 e 4, disponível em https://clip.unl.pt/objecto?oid=78643&oin=Intel-Pentium-Architecture-and-Reference-Manual.pdf

[Art_of_Assembly] Randal Hyde, **The Art of Assembly Language**, capítulo 6, No Starch Press, 2003,
disponível em https://clip.unl.pt/objecto?oid=78919&oin=aoa.pdf

[NASM_DOC] The NASM Development Team, **NASM – The Netwide Assembler version 2.06rc1**, 2008,
disponível em https://clip.unl.pt/objecto?oid=78642&oin=nasmdoc.pdf