***Desempenho***

Depende de:

* Algoritmo: que afeta o número de instruções e o CPI
* Linguagem: que afeta o número de instruções e o CPI
* Compilador: que afeta o número de instruções e o CPI
* ISA: que afeta o número de instruções e o CPI e o período do Clock

No compilador
- Looping unrolling:
Replicar o corpo do laço para reduzir o número de interações.

- Procedure inline:
Substituir chamadas a procedimentos pelo corpo do procedimento, evitando overhead.

Otimização Local/Global

-Locais (blocos simples individualmente)

Branchs, expressões, jumps

-Globais(multiplos blocos simples espalhados pelo código)

-Técnicas:
1. Strength Reduction

Substituir as operações mais complexas por mais simples.

2. Dead code elimination

Eliminar código que não ageta o resultado no final do programa.

3. Code Motion

Tirar dos laços trechos que calculam sempre o mesmo valor.

Meios para avaliar o desempenho
 - Desempenho relativo = TEMPa / TEMPb , TEMPa é 2.0 vezes mais rápido que TEMPb.
 - Tipos de medidas:
 1 - Elapsed Time = Todo o processo, junto com entrada e saida.
 2 - CPU Time = só a CPU
 - Tempo de CPU = TempoClock/Frequência
 \*Pra melhorar, ou diminiu o clock ou aumenta a frequência.
 - CPI (ciclos por instrução)
 Tempo de CPU = Nº instrução\*CPI/Frequencia

***Exceções***

Na ocorrência de uma exceção, o endereço da instrução que causou a exceção deverá ser salvo no registrador **EPC** a ser inserido na unidade de processamento e o **PC** será carregado, posteriormente, com o valor do endereço da rotina de tratamento. O byte menos significativo do endereço da rotina de tratamento está armazenado nos seguintes endereços de memória: **253** (opcode inexistente) e **254** (overflow) e **255**(divisão por zero).

Exceções são eventos não esperados que podem ocorrer dentro do processador.

Lembrando que as interrupções podem ocorrer tanto de dentro quanto de fora do processador.
Consegue dar exemplos de interrupções internas e externas?
>> Interna: Overflow
>> Externa: Dispositivos de E/S

***MIPS***

Sabe dizer os principais registradores?
s0 - s9 = propósito geral
t0 - t9 = temporário
v0, v1 = retorno
a0 - a3 = parâmetro
sp = pilha
gp = aponta para área de memória reservada para as varíveis estáticas.
reg30 = registrador de retorno erro para exceções
at = usado pelo assembler para ampliar o range de um imediato

*Explicação de GP:

POSIÇÃO DA MEMÓRIA INSTRUÇÃO
80000 <- GP add t0, t0, t1
80004 <- GP AGORA TÁ AQUI load t2, 0(t8)
80008 <- E AGORA TÁ AQUI ...*
***IMPORTANTE!!!*** *Quando se faz o lui $reg, immediate, os 16 bits mais significativos do registrador serão salvos com o conteúdo de immediate. Agora, caso se queira depois mudar os 16 bits menos significativos, é aconselhável que se use a instrução “ori” ao invés de “addi”, pois addi leva em conta o sinal...*

MIPS possui uma ISA RISC, bastante utilizada em sistemas embarcados
Desenvolvido no começo de 80

Princípios do MIPS
Simplicidade é favorecida pela regularidade:
 \*Instruções de tamanho fixo
 \*Poucos formatos de instruções
 \*Opcode sempre utiliza os primeiros 6 bits

Quanto menor, mais rápido
 \*Repertório de instruções limitados
 \*Quantidade de registradores limitados
 \*Número reduzido de modos de endereçamento

Torne rápido o caso mais comum
 \*Existência de instruções que contém operandos

Bom projeto requer boas escolhas (compromissos)
 \*Diferentes formatos de instruções complica a decodificação,
 contudo permite instruções de tamanho fixo.

MIPS possui 3 formatos de instrução, todos com 32 bits
R:
op rs rt rd shamt funct

I:
op rs rt immediate

J:
op jump target

Todas as instruções aritméticas possuem 3 operandos, e cada instrução aritmética faz
apenas uma operação
       add a, b, c => a = b + c

Para melhorar o desempenho, os operandos de uma instrução aritmética devem estar nos
registradores (acesso mais rápido em relação à memória)

Operações Lógicas x Operações Aritméticas
A primeira simplesmente insere um 0 no começo ou no final da sequência, enquanto o
segundo mantém o sinal do número.

Instruções de deslocamento tem 3 operandos, ex.:
sll a, b, 3; a = b << 3
srl a, b, 4; a = b >> 4
Possui formato R, rs não é utilizado, enquanto rt tem o operando e o rd o registrador destino, shamt a quantidade de deslocamento. Funct é a função que estende o opcode.

Todos os registradores no MIPS possuem 32 bits (palavra no MIPS)
Número de registradores reduzido (32). Número grande de registradores pode penalizar
desempenho.
Causa impacto no tamanho da instrução.

$s0...$s7 - geralmente para variáveis do programa (16 a 23)
$t0...$t9 - para valores temporários (8 a 15 e 24-25)

MIPS é Big Endian (byte mais significante tem menor endereço na memória)
Memória endereçada por byte (8 bits)
Memória é vista como uma sequência de palavras de 32 bits

Acesso a registradores é mais rápido, pois utilização da memória requer loads e stores, o que implica em mais instruções a serem executadas. O compilador deve, portanto, maximizar a utilização de registradores (a otimização de registradores é importante!) .

No formato de instrução I, o rs contém o endereço base e o rt, o endereço destino, para instruções do tipo load e store. Para instruções do tipo addi, o rs representa o registrador fonte.

Para tornar rápido o caso mais comum, o MIPS oferece instruções onde uma constante está embutida na própria instrução.

Desvios no MIPS
 Branch (desvio condicional)
 Jump (desvio incondicional)

Branch: (formato I)
 *3 operandos (fonte 1, fonte 2, label de instrução)
 ex.: beq a, b, L1; se (a == b) desvie para L1
 beq rs, rt, deslocamento; (PC = PC + deslocamento\*4) se reg1 ==* *reg2*

 O operando relativo ao label corresponde na verdade ao
 deslocamento em relação ao endereço da instrução contida
 no PC (já incrementado de 4!)

Jump: (formato J)
 j L2; desvie para L2
 O operando relativo ao label neste caso corresponde na verdade ao
 endereço da instrução a ser executada

Jump And Link: (formato J)
 jal L3; desvie para L3 e salve o endereço do PC atual + 4 em $ra (31)

Jump Register: (formato R)
 jr $ra; desvie para o endereço contido no registrador ra
 *op rs rt rd shamt funct
 0 $ra 0 0 0 8*
\*O assembler calculará o endereço dos labels

Set Less Than:
 *slt rd, rs, rt; if (rs < rt) rd = 1, else rd = 0*

Subrotinas:
São utilizadas para estruturar o programa, pois facilita o entendimento e aumenta o reuso do código (ex.: procedimentos, funções e métodos). Na chamada de subrotina, o programa executará as instruções contidas nela e após o término da mesma o computador deverá executar instrução seguinte à chamada de subrotina:

 *6 etapas:
 (Caller = quem chama, callee = rotina que é chamada)
 1. Caller coloca argumentos em um lugar onde o callee pode acessá-los
 2. Caller transfere controle para callee
 3. Callee adquire recursos de armazenamento necessários
 4. Callee executa suas instruções
 5. Callee, quando é o caso, coloca o valor do resultado em um lugar onde
 o caller possa acessá-lo
 6. Callee retorna controle ao caller*
 No MIPS, os registrados $a0 a $a3 (4 a 7) são utilizados em geral para passagem de argumentos, mas quando não são suficientes, a pilha é utilizada. Neste caso, a subrotina utilizará registradores cujos conteúdos foram salvos na pilha e terminada a subrotina os valores antigos dos registradores são restaurados. $v0 e $v1 (2 a 3) são normalmente usados como registradores para valores retornados.

Utiliza-se parte da memória como pilha, que cresce do maior para o menor endereço. O registrador $sp (29) guarda o endereço do topo.

 *Push:
 $sp = $sp - 4
 ex.: addi sp, sp, -4 (alocando espaço na pilha)

 Pop:
 $sp = $sp + 4
 ex.: addi, sp, sp, 4 (liberando espaço na pilha)*

Subrotina Aninhada
 Caller que chama irá armazenar na pilha o seu endereço de retorno e os
 registradores que irá utilizar após o término da subrotina chamada

Exemplo:
*int fatorial (int n) {
    if (n<1)return 1;
    else return n \* fatorial (n - 1);
}

FACT: #Função fatorial
#Guardar endereço de retorno e argumento($a0)
addi sp,sp,-8
sw ra, 4(sp)
sw a0, 0(sp)

#Teste n<1
slti t0,a0,1
beq t0,zero,L1

#if->true
addi v0,zero,1 #return 1
addi sp,sp,8 #Ajustando ponteiro da pilha, pois os valores salvos não serão nescessários
jr ra #Retorno para a última chamada recursiva

L1:
#Chamada recursiva
addi a0,a0,-1 #n--;
jal FACT
#Retorno da chamada recursiva
#Recuperando dados salvos em pilha
lw a0,0(sp)
lw ra,4(sp)
addi sp,sp,8
mult v0,a0 # n \* fatorial (n - 1)
mflo v0 # n = n \* fatorial (n - 1)
jr ra*

Organização da Memória no MIPS
 Segmento de Texto: código
 Static Data: variáveis globais
 No MIPS, o registrador $gp guarda o início deste segmento
 Dados Dinâmicos: heap
 (ex.: malloc em C, new em Java)
 Pilha (Stack)
 Variáveis locais
 Registradores

Comparação MIPS vs Outras Arquiteturas
Endereço de retorno:
No MIPS, se dá no registrador (melhor desempenho), enquanto em outras estruturas é na memória.

Acesso à Pilha:
No MIPS, é por meio de instruções lw e sw (menor complexidade na implementação), enquanto nas outras é por meio de instruções adicionais.

Chamas aninhadas ou recursivas:
No MIPS, é implementada pelo compilador (compilador mais complexo), wnquanto em outras há um suporte direto da máquina.

Modos de Endereçamento:
Se refere às maneiras em que instruções de uma arquitetura especificam a localização do operando (onde e como pode ser acessado)

No MIPS, operandos podem estar em registradores, na memória ou na própria instrução.

Endereçamento de Registrador
O operando está em um registrador e a instrução contém o número do Registrador.
 *Ex.: add R3 (00011), R3 (00011), R7 (00111)*
Endereçamento Base
 *Ex.:* *Endereço inicial + deslocamento*

Endereçamento imediato
 O operando é especificado na instrução
 *Ex.: addi R1 (00010), R2 (00001), 12 (000...01100)*

Endereçamento (Pseudo)Direto
*Ex.: Instrução de Desvio Incondicional o (pseudo)endereço da próxima instrução é especificado na própria instrução. Os 4 bits mais significativos do PC são concatenados ao endereço especificado multiplicado por 4.* *j endereco
 PC = conc(PC(4 bits), endereco\*4) \*\*(endender melhor)*
Endereçamento Relativo a PC
 Ex.: Instrução de Desvio Condicional
 o número de instruções a serem puladas a partir da
 instrução é especificado na instrução

 beq R1, R2, desl => PC = PC + desl\*4

***Ponteiros no MIPS***

Em questão de otimização/desempenho, sempre prefira usar ponteiros ao invés de indexar a array normalmente... disseram que só implementando a questão com ponteiros o professor já considerou uma otimização total...

 &v[i] = (v+i)
 Versão de clear usando ponteiros:

 *int clear (int \*array, int size) {
 int \*p;
 for (p = array, p < array+size; p++) {
 \*p = 0;
 }
 }

 addi $t0, $a0, 0          # p = &array[0], iniciando o ponteiro com o valor passado como arg
 sll $t1, $a1, 2             # $t1 = size\*4 , calculando o descolamento máximo
 add $t2, $a0, $t1       #Calculando a última posição do array (array+size)
 L2: sw $zero, 0($t0)  #\*p = 0
 addi $t0, $t0, 4           #p++
 slt $t3, $t0, $t2           #(p<array+size) ? t3=1:t3=0
 bne $t3, $zero, L2      #Se p<array+size, realizar loop de novo; se não, encerrá-lo*

***Informações de um Módulo Objeto***

Header
 Descreve conteúdo do módulo
Segmento de texto
 Instruções traduzidas
Segmento de dados estáticos
 Informações de relocação Identifica instruções e dados que dependem de endereços absolutos quando o programa for carregado na memória.

Tabela de Símbolos
 Associa labels com endereços podem existir ainda labels indefinidos (não associados a nenhum endereço)
 *ex.: referências para rotinas externas*

Ligando módulos de objetos
 Um programa geralmente é composto por diferentes módulos, que podem ser compilados separadamente. Uma mudança em uma rotina implica em compilar apenas um módulo do sistema.

Linker é o programa que combina módulos de objetos compilados independentemente.

Produz um executável, que estabelece como os diferentes módulos devem ser organizados na memória e determina os endereços dos labels definidos (através da tabela de símbolos e informações de relocação)

Abordagem tradicional de ligação é estática (códigos das rotinas dos módulos
externos chamados no código do módulo principal são incorporados ao executável, mesmo se não forem executados.

Se os módulos externos forem modificados, o executável continua com o módulo antigo.

**Solução: Ligação Dinâmica (DLLs)**
Fazer ligação apenas quando a rotina precisar ser executada. Requer que o
programa e rotinas armazenem informações extras (nome das rotinas, local
das rotinas).
***descrição:****“DLLs são partes de programa que são indexados dinamicamente, isso quer dizer, que moficicações realizadas nelas não resultarão em recompilação do código completo, e sim apenas da DLL.”*

Loader é o responsável por carregar um programa na memória, e ele faz parte
do sistema operacional.
*Passos:
 1. Lê header para determinar tamanhos dos segmentos
 2. Cria espaço na memória
 3. Copia texto e inicializa dados na memória
 4. Coloca argumentos da main na pilha
 5. Inicializa registradores ($sp, $gp...)
 6. Pula para a rotina de início
 Copia argumentos para $a0, ... e chama main, e quando main retorna, ele sai do programa.*
*Em Java:*
 JVM faz a ligação (link) do programa principal com os métodos das bibliotecas (pacotes) padrão, e também funciona como um loader. Também pode chamar o JIT (Just In Time) para melhorar o desempenho, pois este identifica os métodos mais lentos e usados e compila-os para linguagem de máquina e salva o código compilando, provendo um melhor desempenho em futuras execuções.

 Os operandos ficam em uma pilha, e não em registradores. Bytecodes (linguagem entendida pela JVM) variam de tamanho (1 a 5 bytes)

 JVM garante execução segura dos bytecodes (ex.: instruções envolvendo arrays não depassam limites do array).

Algumas conclusões sobre o x86
 *Instruções complexas tornam difícil a implementação
 Hardware traduz instruções para micro-operações mais simples
 Microengine similar ao RISC
 Mercado torna o processador economicamente viável*

Desempenho comparável ao RISC
 *Compiladores evitam instruções complexas*

Compatibilidade amarra projeto do processador
 *Elegância técnica != sucesso no mercado*

**Implementação Monociclo**
 Vantagens:
 Simplicidade de implementação (datapath e controle)

 Desvantagens:
 Custo de hardware (duplicação de componentes devido a
 restrição de um ciclo para o processamento de instrução)

 Desempenho:
 Ciclo de clock longo para comportar instrução mais lenta
 Instruções que demandam menos tempo deixam CPU ociosa
 Implementação pouco eficiente

**Implementação Multiciclo**
 Veja o seu projeto!

***Questões***

**Questão de prova, 2011.2:
1) Explique como funciona o endereçamento no MIPS, suas pilhas e como uma rotina pode ser chamada através dela (detalhando os registradores usados no processo).**
A pilha funciona da seguinte maneira: Você possui um registrador ($sp) que aponta para o topo da pilha (que cresce do maior endereço na memória para o menor). Para alocar espaço na pilha, você decresce o valor de $sp (por estar decrescendo sua posição na memória, você estará alocando espaço). Para desalocar espaço na pilha, você simplesmente acresce o valor de $sp de 4\*i, sendo i quantos words você quer desalocar. Pilhas são bastante úteis nas chamadas de rotinas, por dois motivos principais:
     1) Quando não há registradores suficientes para passar o argumento de uma rotina, você pode simplesmente guardar todos os registradores que está a usar atualmente na pilha e usar os registradores agora vagos para o procedimento
     2) Dentro de um procedimento, você pode usar a pilha para realizar rotinas aninhadas (recursivas, por exemplo), pois você pode armazenar na mesma os valores que está a usar e o endereço da rotina atual, para que mais tarde possa-se voltar a ela.
No MIPS, dois registradores são usados como registradores de retorno ($v0 e $v1). $v0 é normalmente usado quando o valor de retorno cabe em um inteiro de 32 bits. Quando este não é o caso, usa-se o registrador $v1 para armazenar a parte mais alta do valor de retorno. Estes registradores também são usados caso se queira retornar um valor de precisão dupla.

 **2) Qual a vantagem de multiciclo?**
Dentre as principais vantagens podemos citar o fato de que gera um grande reuso do hardware, simplicidade do projeto, facilita a adição de novas funções. Mais eficiente que o monocliclo, pois nesse intruções menores levam menos tempo para executar.

**3) Questões recorrentes:**
1) Comparação de MIPS com outras arquiteturas ou apenas detalhamento do MIPS.
2) Adicionar novas instruções no seu projeto e dizer como ele ficará e o que deverá ser mudado.
3) Dado um código, transforme ele pra assembly.
4) (segundo fontes, o mais trabalhoso) Dado um código, otimize ele o máximo possível e agora detalhe cada uma das instruções que você usou de acordo com a sua arquitetura (geralmente se usa mais de 20 instruções, então...)

**4) Sabe explicar pq um código compilado em um processador AMD funciona em um INTEL?Porque eles tem a mesma ISA?**Não a mesma ISA, mas ISAs compatíveis.
Eles tentam adaptar a compatibilidade principalmente por causa do mercado. Mudar o repertório de instruções meio que bifurcaria a sapiência [...] (Eu colocaria, mantem a instruções básicas), ok?
Nem todo código AMD roda em um INTEL. Instruções específicas de uma geração podem não rodar.