Leandro Kümmel Tria Mendes	033910	Alexandre Nobuo Kunieda	080523
Fernando Teixeira Barros	085858	Marcus Felipe Botacin	103338

Grupo 11 - Medidas no processador Mips

Introdução

Em prosseguimento aos estudos desenvolvidos no trabalho 1, analisa-se agora um simulador de processador MIPS comportamental, ou seja, que não considera nenhuma informação de temporização e de estruturas internas, objetivando verificar os efeitos de implementação destas no mesmo. Consideraremos um *pipeline* de 5 estágios, *branch prediction e cache*, e realizaremos uma série de benchmarks afim de verificar qualitativamente os efeitos dos mecanismos implementados.

Objetivos

Neste projeto, desejamos avaliar qual o impacto das estruturas arquiteturais e das decisões de projeto do processador no desempenho do mesmo, especificamente em relação à implementação de *branch prediction* e *caching* no processador.

Métodos

Será considerado um processador MIPS, como visto em aula, que executa uma instrução por ciclo, com 5 estágios de pipeline, modelado na plataforma ArchC.

Estudou-se cada tipo de *hazard* que pode acontecer. Entre os *Data Hazads*, optamos por analisar as condições de hazard load-use, uma vez que não é possível realizar forward, obrigando a inserção de um stall na pipeline.

Verificou-se também a necessidade de avaliar a ação do mecanismo de branch prediction sobre a execução de diferentes programas e a influência dos mecanismos de cache, através da medida do desempenho para diferentes configurações de cache.

O Procedimento se deu, inicialmente, através da implementação dos mecanismos descritos acima, ocorrendo, posteriormente, a avaliação dos resultados. Para este fim, executou-se uma série de benchmarks (mibench, site do archc), cujos resultados seguem na seção correspondente.

Implementação

- **Implementação da Representação de um pipeline**

Dado que o simulador é comportamental, não temos informações relativas ao pipeline existente no processador mips, sendo que estas informações são fundamentais para determinar a ocorrência de eventos e para podermos alocar os recursos a serem programados no estágio correto, de modo a obter resultados consistentes.

Deste modo, iniciou-se o trabalho pela implementação de estruturas que representassem o pipeline: `control_t IF, ID, EX, MEM, WB;`

Adicionou-se uma função chamada ***storeInstruction*** de modo que esta é a responsável por controlar as instruções correntes no pipeline: `void storeInstruction(int ins_id, int rs, int rt, int`

rd) {
WB = MEM; MEM = EX; EX = ID; ID = IF;

Nota-se que esta verifica o que deve ser armazenado para cada instrução:

```
} else if (ins_id >= 21 && ins_id <= 45) {
       // R-type
       //  21-45
       IF = control_t(rs, rt, rd);
} else if (ins_id >= 48 && ins_id <= 49) {
       // jump
       //  48-49
       IF = control_t(rs);
} else {
       // jump to imediate
       //  46-47
       // syscall, instr_break
       //  58-59
       IF = control_t();
```

Ela também é responsável por definir as flags, tais como o uso de memória, explicado abaixo:

```
       IF.memread = useMemory(ins_id);
       IF.ins_id = ins_id;
```

- **Implementação do uso de memória.**

Dentre as muitas flags gerenciadas pela unidade de controle, uma delas é responsável pelo sinal de *MemRead*. Visando implementar este sinal, a ser usado na detecção de hazards, como explicado abaixo, implementou -se a função que segue:

```
bool useMemory(unsigned ins_id) {
       return (ins_id >= 1 && ins_id <= 7); // intervalo dos loads
}
```

- **Hazards**

A detecção de load-use hazards, se dá com a seguinte condição:
ID/EX.MemRead And (ID/EX.RegisterRt=IF/ID.RegisterRs or ID/EX.RegisterRt=IF/ID.RegisterRt)
tal qual mostrado no slide 77 ref: Chap 4, do livro de referência.
Assim, implementou-se a função:

```
checkHazard(ac_pc.read());

if (ID.memread &&
    (ID.rt == IF.rs || ID.rt == IF.rt)) {
       hazard_count_by_type[DATA_HAZARD]++;
       hz = true;
```

```
}

if (hz) hazard_count++;
```

Onde hazard_count é uma variavel acumuladora que visa contar o numero de hazards. Este valor é impresso em void mips1::stop(int status)

- **Branch Prediction**

O branch predictor de um processador visa evitar stalls na pipeline por impossibilidade do processador conhecer a próxima instrução a ser executada. Isso acontece em operações de branch, em que uma comparação é feita antes da decisão do salto a ser executado. Na arquitetura do processador MIPS, só conhecemos o resultado do branch ao final da etapa ID, então teríamos um stall por operação de branch.

Uma dificuldade do branch prediction é que, como não podemos aguardas o estágio ID, seu processamento deve ser feito na etapa IF, em que ainda não conhecemos a instrução a ser executada nem os parâmetros da instrução (no caso do branch, o endereço de destino do salto).

Implementamos um branch predictor dinâmico de 2 bits em uma hash no PC/4 (dividido por 4 porque os PC são sempre múltiplos de 4). O branch predictor consiste em uma máquina de estados de 4 estados (daí os 2 bits) em que metade dos estados representa que a instrução é um branch e que o salto acontece; e a outra metade representa instruções que não são branch ou branches cujo salto não deve ocorrer. No caso em que o branch predictor acha que o salto deve ocorrer, há também um endereço de destino armazenado para realizá-lo.

A implementação consiste essencialmente em uma classe do branch predictor chamada branch_pred_t, e uma função checkBranchPred() que recebe o endereço da instrução executada, se ocorreu um salto na instrução e, se sim, o endereço de destino do salto e confere se o branch predictor acertaria sua previsão ou não.

Adicionamos uma chamada à função checkBranchPred() para cada instrução e contamos o número de predições incorretas.

- **Implementação Adicional**

Realizou-se também a verificação da influência do mecanismo de forwarding nos resultados. Assim apresentamos, adicionalmente, o número de stalls que ocorreriam na ausência do mecanismo de forwarding no processador.

Para a identificação dos hazards realizou-se a verificação tal qual a exibida no slide **69** do livro-referência.

- **Cache:**

Foram considerados dois tipo de cache, de dados (DC) e de intruções (IC). Ambas têm a política de LU (“*Least Used*”) para remoção de elementos da cache, caso não tenha mais espaços livres na mesma. Já a política de escrita, de memória, utilizada denomina-se “*write-back*” (ou WB), e foi utilizado um “*dirty-bit*” para cada entrada do cache de dados.

A configuração da tabela de cache pode ser alterada na *library.h*, onde é possível setar uma configuração DM (“direct map”) ou “Fully associative” , N-way, em nosso caso, utilizamos a seguinte

configuração:

IC com 512 linhas e 16-way e DC com 1024 linhas e 16-way.

Alguns trechos de códigos interessantes:

- Estrutura da IC e DC: Nota-se o *dirty-bit* para a cache de dados, no caso da instrução *store*.

```
typedef struct  dcache {
  unsigned int addr[DC_NWAY];
  int use[DC_NWAY];//num de vezes utilizado
  bool dirty[DC_NWAY];//dirty-bit , write back policy
};
typedef struct  icache {
  unsigned int pc[IC_NWAY];
  int use[IC_NWAY];//num de vezes utilizado
};
```

- Caso HIT: Em caso e hit o contador use é incrementado, cada posição da cache contém um. Na política LU, a posição com menor valor do contador é removida. Nota-se que no caso DM o i é constante em i=addr%(DCACHE_LEN), já o fully-association o i varia de 0 a, no máximo, DCACHE_LEN, ou até achar o registro.

```
if(store)//caso seja funcao de store
  DATA_CACHE[i].dirty[j]=true;//marcar dirty-bit
  DATA_CACHE[i].use[j]++;//hit
if(INSTR_CACHE[i].pc[j]==pc)
  INSTR_CACHE[i].use[j]++;//hit
```

- Política LU:

```
if(DATA_CACHE[i].use[j]<min){
  min=DATA_CACHE[i].use[j];
  emin=i;
  jmin=j;
}
```

Como a implementação de todo o projeto, bem como da cache, queremos analisar o desempenho então, vale notar que para DC é salvo endereço da memória (e não, também, o dado), denominado DC_ADDR, essa é atualizada a cada operação na memória, seja *load* ou *store*. Já para a IC, escrevemos o valor de PC-4 uma vez que, no MIPS todas as instruções têm 32bits e é alinhado em 4 bytes, a instrução é armazanada na IC depois de ser decodificada e o PC já ser incrementado por 4.

Resultados

Para verificar o desempenho executou-se os seguintes benchmarks do grupo Mibench.

- QuickSort (Large & Small)
- Susan Small (Corner & Edge & Smoothing)
- Basic Math Small
- Bit Count (Large & Small)
- CRC 32 (Large & Small)

- ADPCM Small (Encode & Decode)
- FFT Small (Common & Inv)
- GSM Small (Encode & Decode)
- Djikstra (Large & Small)
- Patricia (Large & Small)
- Rijndael Small (encode & Decode)
- SHA (Large & Small)
- JPEG (Large & Small) (Encode & Decode)

Os Resultados são apresentados em tabela Anexa.

Conclusões

Neste trabalho, pudemos verificar a influência de diversos mecanismos internos do processador em seu desempenho. Destaca-se a grande quantidade e a diversidade de testes realizados, de forma que estes compõem uma base sólida de análise.

Pode-se verificar, através dos comparativos, que o mecanismo de forward é responsável por um ganho substancial de desempenho no pipeline. Ademais, pode-se verificar também que os casos de *data hazard* se resumem aos casos de *load-use*, graças a este mecanismo.

Verificou-se também o ganho de desempenho obtido com a presença do mecanismo de *predição de branch*, o qual consideramos significativo, embora em menor proporção que o mecanismo de *forwarding*.

Também analisou-se o mecanismo de cache do processador, este dividido em *Instruction Cache* e *Data Cache*. O primeiro apresentou, tal qual esperado pelo modelo teórico, baixa taxa de *miss*, sendo que os casos de falha causados principalmente por instruções de *branch*, ao passo em que o segundo tem desempenho variado de acordo com a configuração e as politicas utilizadas, tal qual os casos expostos nos diversos testes realizados.

Cabe, ainda, destacar a grande influência do compilador nestes resultados. Nos testes realizados, pequenos programas, não otimizados, apresentavam muito mais condições de *hazard* do que nos benchmarks reais.

Por fim, dado os resultados apresentados, conclui-se que o grupo obteve sucesso em realizar as análises propostas.

Anexo:

Tabela de resultados dos benchmarks