UNIVERSIDADE ESTADUAL DE MARINGÁ
DEPARTAMENTO DE INFORMÁTICA
Curso: Ciência da Computação
Disciplina: 6887/02 – Arquitetura e Organização de Computadores

Relatório do Segundo Trabalho de Arquitetura e Organização de Computadores

Alunos:
Gabriel Crespo Biscaia 118928
Vitor Augusto Greff 118926

Professor:
Felippe Fernandes da Silva

Maringá, Abril, 2022

Introdução

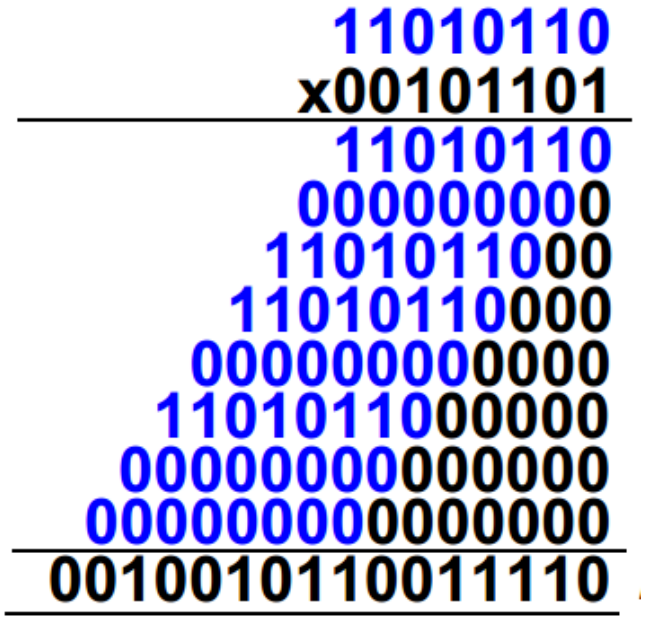
O trabalho foi implementado na linguagem **java**, ela foi escolhida pois nesse semestre fomos introduzidos a ela juntamente com **POO** (Programação Orientada a Objeto) e tendo isso em mente, fazia sentido usá-la pois cada parte do processador seria uma classe, ademais todos os outros trabalhos e atividades foram feitas nessa linguagem.

Desenvolvimento

Tínhamos como princípio pegar os principais registradores da **CPU** e fazer uma classe para cada um, com isso temos a classe principal cujo o nome é “Cpu.java” que faz a ligação entre as demais classes.

Começamos pelas operações de soma e subtração cujo quais foram fáceis de serem implementadas. Fizemos uma série de comparações lógicas com resultados pré estabelecidos, além disso, utilizamos a variável booleana **“vai_um”** para representar os bits que seriam utilizados em comparações posteriores. O seu valor será atualizado sempre que necessário tendo como valor inicial “**false’**.

Partindo para a multiplicação, nos agarramos em um exemplo prática, sendo a seguinte imagem nossa principal referência visual:



Exemplo de multiplicação binária.

Com a imagem de referência, tratamos a multiplicação como uma série de somas parciais que seriam alocadas em uma variável auxiliar. Foi um processo difícil porém com alguns ajustes, principalmente quanto ao tamanho dos resultados, conseguimos realizar a multiplicação até o máximo permitido por dois operandos de 6 bits cada. Aproveitamos para fazer com que uma condição, se cumprida, chamasse uma função para informar um overflow. Essa condição seria verificada antes do retorno dos resultados das operações de soma e multiplicação.

Quanto à divisão, pela falta de tempo necessário para pensar em como implementá-la, acabamos por decidir fazer a conversão para decimal e retornar o resultado como binário. Acreditamos se tratar de uma operação complexa demais para lidar com o pouco de tempo que nos restava, ainda mais com outras tarefas tão importantes que deveriam ser realizadas.

As principais classes de nosso trabalho são a “Cpu.java” e a “Ula.java”, pois a primeira faz conexão de todas as outras classes java, e a segunda tem como objetivo fazer as operações em binário, que no nosso caso deixamos os números em formato de Strings.

A segunda parte do trabalho fez nós implementarmos o Pipeline no código apresentado acima, fizemos isso adicionando uma função na “Cpu.java” chamada “pipeline()”. Para o implemento da mesma, separamos a função principal “cicloDeBuscaExecucao()” em outras subfunções para que assim cada processo do ciclo de instrução pudesse ser executado em clock diferente, caso ao contrário todos os processos do ciclo de instrução ficariam no mesmo clock, impossibilitando a implementação do pipeline.

O pipeline por sua natureza é divido em 6 processos, dentre eles:

1 - Buscar instrução;
2 - Decodar instrução;
3 - Calcular operandos;
4 - Buscar operação;
5 - Executar instrução;
6 - Escrever resultado.

Contudo, nós não utilizamos os itens 3 e 4. Não utilizamos o “calcular operandos” pois do jeito que foi modelado o trabalho a parte de calcular operandos já acontece no item 5 (Executar instrução), e não utilizamos o item 4 pois a “busca operação" também já acontece no switch que nós fazemos, ficaria muito redundante fazer uma função para retornar um operando a partir de um opcode e depois utilizá-lo em outra função, com isso o item 5 fica uma mescla do 3, 4 e 5.

Com isso as tarefas são divididas da seguinte forma no nosso pipeline:

1 - Buscar operação (fi);
2 - Decodar instrução (di);
3 - Executar instrução (ei);
4 - Escrever resultado (wo).

Ademais não aplicamos os “Hazards” em nosso trabalho, apenas as divisões de tarefas acima.

Facilidades

O trabalho em si não apresentou muitos problemas, foi relativamente fácil, e gostamos muito de fazer, pois era lógica desde buscar a palavra na “memoria1.txt” e ir passando de registrador em registrador até ser feita a operação e ser armazenada na “memoria2.txt”.

A soma e subtração foram de fácil implementação comparada a multiplicação e divisão.

O incremento da pipeline teve seu ponto fácil de difícil, como nosso código estava bem baseado/estruturado, para dividir as tarefas precisou apenas separar a função principal da CPU em subfunções para poder colocar as mesmas em tempos diferentes.

Dificuldades

De longe, a parte mais difícil do trabalho foi a implementação das operações em multiplicação e divisão binário, tanto que, na segunda fizemos a conversão de binário para decimal para fazer a operação e posteriormente transformamos o resultado em binário novamente. Fizemos isso pois a operação de divisão possui uma alta complexidade e chegamos a conclusão que tentar implementá-la em binário causaria muitos problemas e provavelmente não conseguiríamos acabar a tempo.

A parte que tivemos que pensar mais para a implementação do pipeline foi como pegaríamos resultados antigos de processo que já foram, para isso criamos alguns vetores com “50” de espaço de limite para armazenar valores antigo, então caso uma função precisasse de algo que já aconteceu ela acessava o resultado armazenado em um vetor no clock.

Por exemplo, a função “escrever resultado” sempre estava um clock atrás da “execução da instrução”, com isso, armazenamos em um vetor todos os resultados da “execução da instrução” no clock[x], quando a função escrever resultado chegasse nesse clock, o resultado já estaria armazenado para ser mostrado na tela.

Como utilizar o programa

Para executar o programa corretamente, é necessário mudar o path em dois lugares do código: O da leitura da “**memoria1**” e o da escrita da “**memoria2**”. Além disso, cada palavra **deve** conter 16 bits e ter um opcode de 1-4 em binário. Portanto, é necessário escrever palavras na **memoria1.txt** para servirem de instrução. Os 4 primeiros bits devem ser o opcode e o restante devem representar os operandos com 6 bits cada.

Caso o opcode seja inválido, uma mensagem será exibida e a instrução será automaticamente ignorada, continuando a execução para a próxima palavra. Caso esteja tudo em ordem, o resultado será armazenado na **memoria2.txt**.

Conclusão

Com isso concluímos que o trabalho em si foi muito divertido de fazer e gerou muito aprendizado, contudo as operações de divisão e multiplicação foram um empecilho para a fluidez do trabalho pois as duas, além de serem muito complexas, geraram muitos bugs que conseguimos contornar na multiplicação, contudo na divisão foi necessário fazer a conversão para decimal.