Лаб. Упражнение No: 2

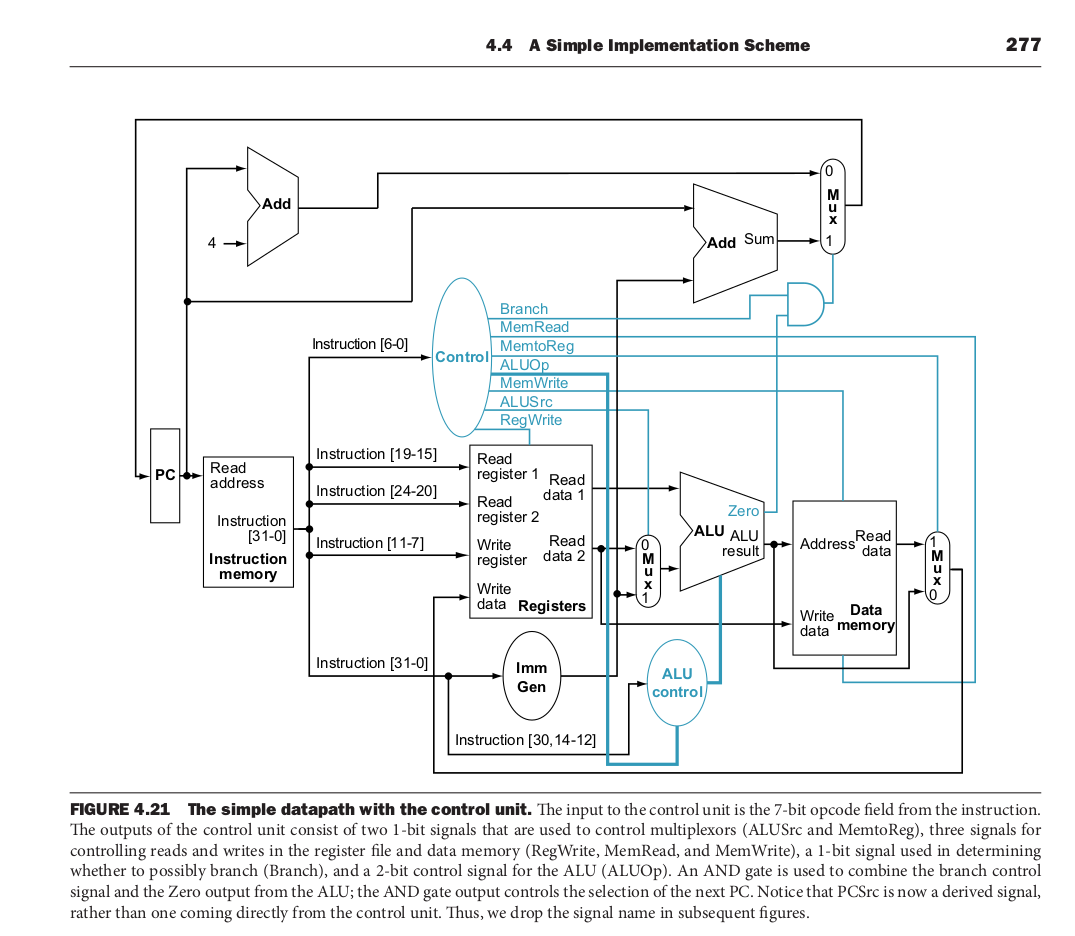
Дисциплина: Компютърни Архитектури

Асистент: Иван Янчев

========================================================================

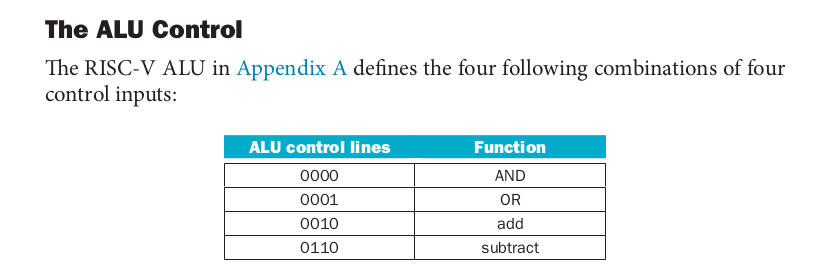
1. **Теория:**

**Control Unit** (Управляващо устройство) – декодиращ елемент задаващ контролните управляващи сигнали за инструкциите спрямо подаденият му вход(OPcode).



Фиг 1.

**ALU control** – четирибитова управляваща шина, подаваща операция на ALU-то. За текущото упражнение ще се използва **подмножество на валидните операции**. На таблицата долу са показани операциите и техните кодове.



Фиг 2.

**Видове управляващи сигнали:**

Branch – Контролен сигнал указващ дали изпълняваната инструкция е инструкция за преход. Този сигнал е подаван с флаговете, генерирани от ALU-то, към логически елемент (на фиг 4.21 - лог. “И”) за правилна обработка на инструкциите за преход.

MemRead – Сигнал указващ на данновата памет, че ще се извърши операция за четене.

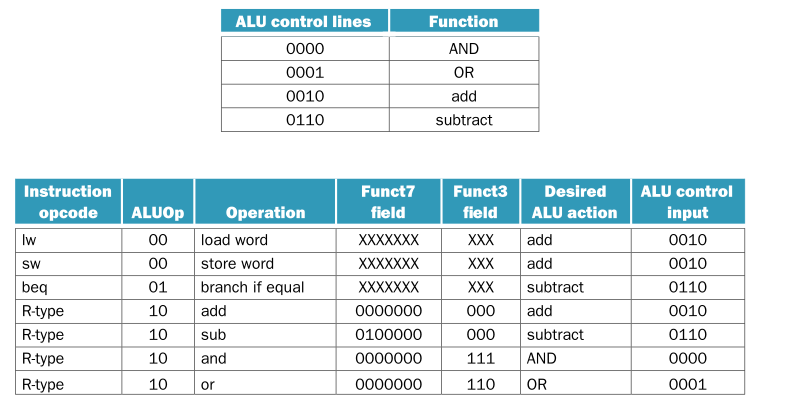
MemtoReg – Управляващ сигнал за мултиплексора, указващ дали данните за записване в регистрите ще се вземат от данновата памет, или от ALU-то.

ALUOp – Управляващ сингал за ALU-то, **00 означава събиране, 01 означава изваждане, при 10 командата се взима от funct3 полето. Ако funct3 полето е с код 000, инструкцията се взима от funct7. funct7 полето е налично само при R-формат инструкции. funct3 не е налично при U и J формати.** Този тип декодиране се нарича многослойно декодиране и е популярна практика при дизайна на компютърни архитектури. С него може да се намали латенцията на управляващото устройство. (фиг 3 показва кодовете на инструкциите, разгледани до сега. “X” означава “don’t care”, X битовете не се използват и следователно тяхната стойност няма значение)

MemWrite - Сигнал разрешаващ писането в данновата памет.

ALUSrc – Управляващ сигнал за мултиплексора, указващ дали вторият операнд на ALU-то да се вземе от immediate полето, или от регистрите.

RegWrite - Сигнал разрешаващ записване в регистровия файл.

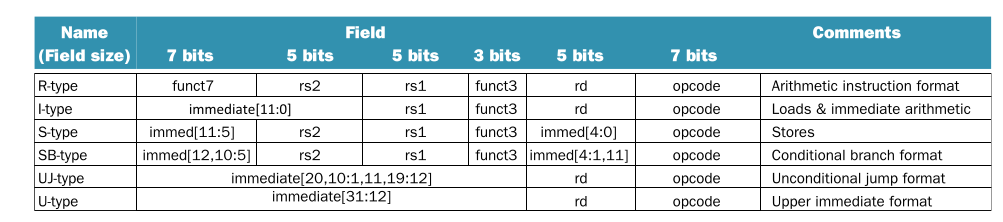


Фиг 3.

**U и UJ формат инструкции**

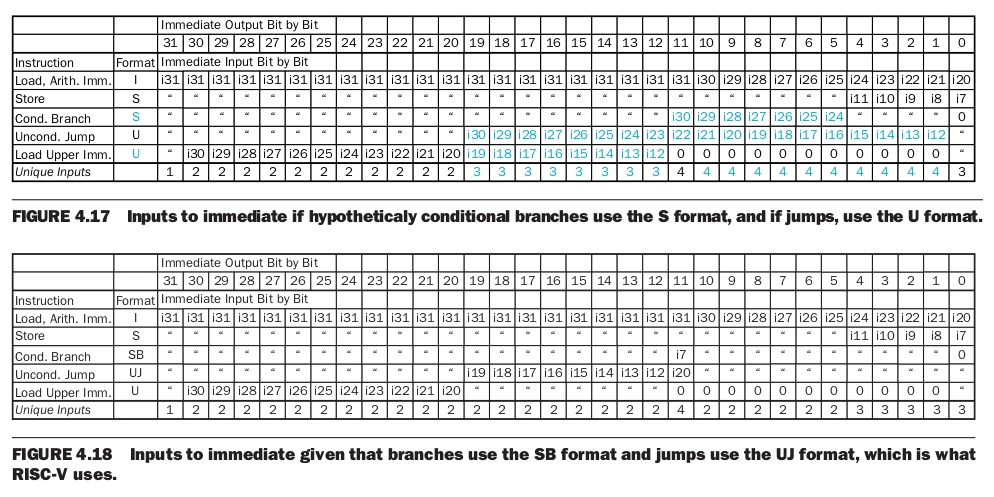
UJ(jump) - използва се за безусловни преходи. Пример: извикване на функция.

U(upper) – използва се за зареждане на горните 20 бита от регистър.



Фиг 4.

**Защо съществуват почти идентични формати -> S и SB, J и UJ?**



Фиг 5.

Нека разгледаме таблици 4.17 и 4.18. **4.17** показва колко различни входа ще получава immediate генератора(imm gen), за всеки от 32-та бита. тези битове трябва да бъдат подадени на съответен мултиплексор за да се избере един, ако SB беше B и UJ беше U. **4.18** показва същото, но при стандартната реализация. “**”**” означава повтаряемост. Нека първата инструкция от таблицата (Load. Arith. Imm) бъде отбелязана с “**1**”, докато последната (Load Upper Imm.) с “**5**”, инструкциите между тях със съответния номер следващ поредността.

Тогава при инструкция **1**(I формат) битове 31-20 са immediate и се копират като най-младшите (11-0) бита в imm gen-а. 11-тия бит се копира в останалите най-старши 20 бита, за да се получи sign extend-а.

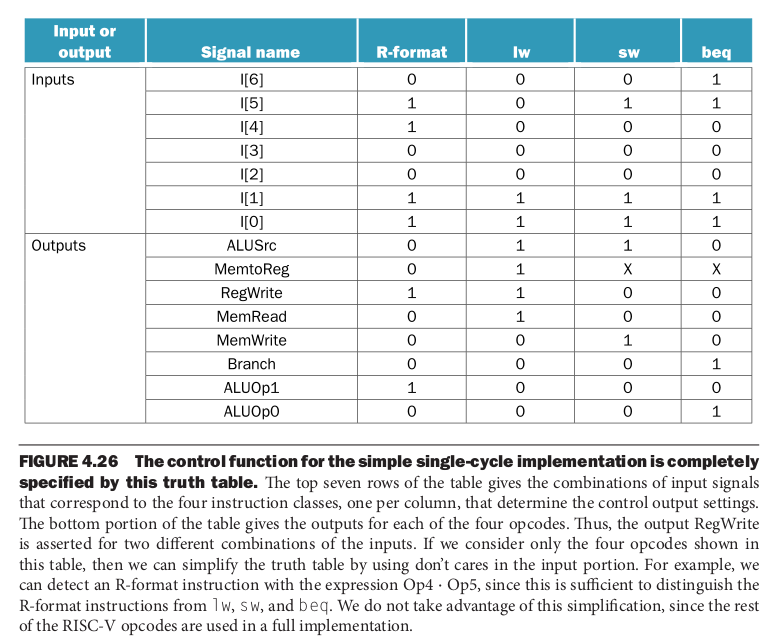
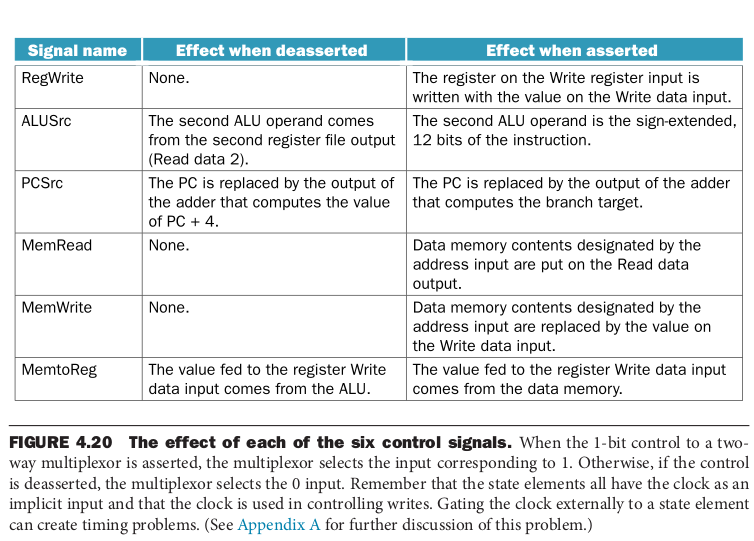
При инструкция **2** (S формат) принципът е същият, но последните 5 бита се взимат от (11-7) от инструкцията. За тях е нужен допълнителен вход на мултиплексора.

При **3** е първата разлика в таблиците. В 4.17 понеже последния бит се shift-left-ва, което разширява ефективния формат 2 пъти( 2 пъти по-голям скок), битовете (11-5) в 4.17 са изместени с един на ляво, което създава допълнително ненужни входове. При използване на SB форматът този проблем е решен.

В 4 и 5 ситуацията е сходна.

С добавянето на SB и UJ нужният хардуер, както и латенцията на елемента(imm gen) намаляват.

1. **Допълнителна Теория:**



1. **За протокола:**

4.1, 4.3, 4.4, 4.5, 4.6 от учебника - стр.370**.**

1. **Линкове**

**Упражнения**: [https://github.com/tu-iyan/Computer-Architecture-2024](упражнения)

**Учебник:** [http://library.lol/main/373C67B0C5E22C9B92B1D8FACDC47E68](учебник)