Implementação sequencial do RiSC16

Este artigo descreve uma implementação sequencial do Ridiculously Simple Computer de 16-bits, uma ISA didática que é baseada no Little Computer (LC-896) desenvolvido por Peter Chen na Universidade de Michigan.

1. Conjunto de instruções RiSC-16

O RiSC-16 é um computador de 16-bits e 8 registradores. Todos os endereços são do tipo shortword (isto é, o endereço 0 corresponde aos dois primeiros bytes da memória principal, o endereço 1 corresponde aos dois segundos bytes da memória principal, etc). Como o conjunto de instruções da arquitetura MIPS, por convenção de hardware, o registrador 0 sempre conterá o valor 0. A maquina impõe isso: ler o registrador 0 sempre retorna 0, independente do que tenha sido escrito nele. Há três formatos de código de instrução e um total de 8 instruções. O conjunto de instruções é dado na tabela abaixo.

Assembly-Code Format		Meaning
`add`	`regA, regB, regC`	R[regA] <- R[regB] + R[regC]
`addi`	`regA, regB, immed`	R[regA] <- R[regB] + immed
`nand`	`regA, regB, regC`	R[regA] <- ~(R[regB] & R[regC])
`lui`	`regA, immed`	R[regA] <- immed & 0xffc0
`sw`	`regA, regB, immed`	R[regA] -> Mem[R[regB] + immed]
`lw`	`regA, regB, immed`	R[regA] <- Mem[R[regB] + immed]

`beq`	`regA, regB, immed`	if (R[regA] == R[regB]) {
		PC <- PC + 1 + immed
		(if label, PC <- label)
		}

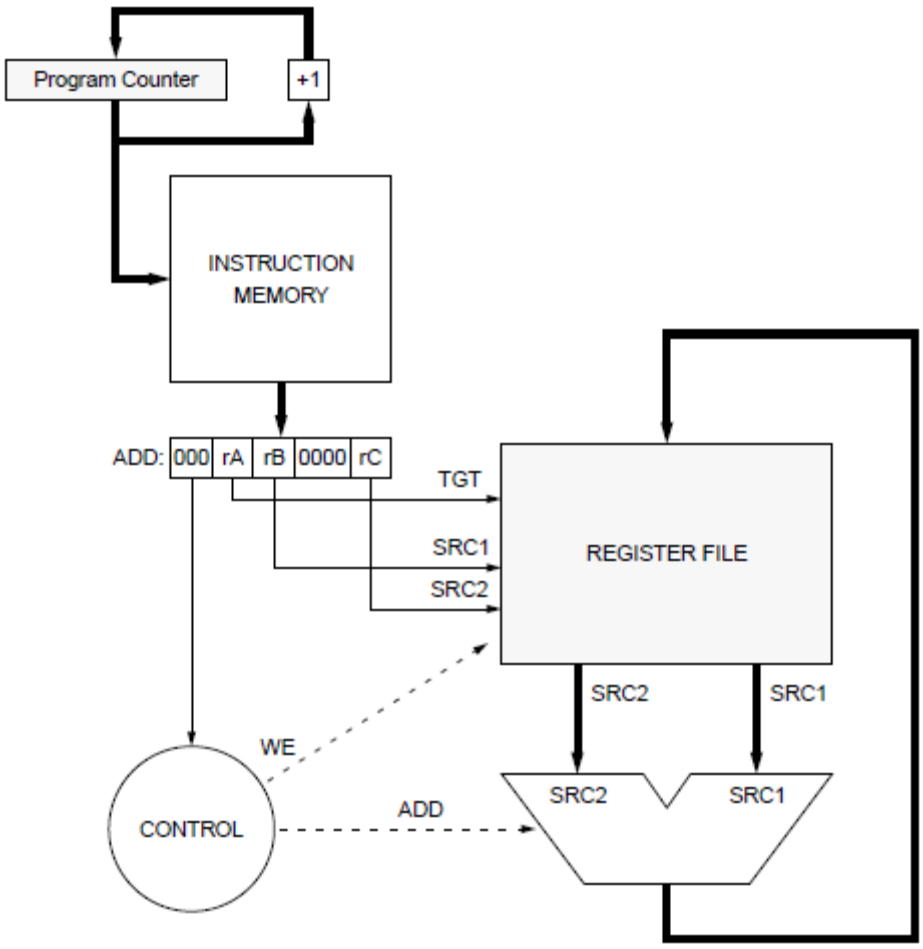
`jalr`	`regA, regB`	PC <- R[regB], R[regA] <- PC + 1
PSEUDO-INSTRUCTIONS:		
`nop`		do nothing
`halt`		stop machine & print state
`lli`	`regA, immed`	R[regA] <- R[regA] + (immed & 0x3f)
`movi`	`regA, immed`	R[regA] <- immed
`.fill`	`immed`	initialized data with value *immed*
`.space`	`immed`	zero-filled data array of size *immed*

O conjunto de instruções é descrito em mais detalhes (incluindo o formato do código de máquina) em *The RiSC-16 Instruction-Set Architecture*.

2. Instrução de Controle e Fluxo de dados no Risc-16

As seguintes figuras ilustram o fluxo de dados para cara tipo de instrução de uma implementação sequencial simples – uma versão reduzida do que foi feito em lógica discreta (isto é, componentes TTL de 4 bits). A última figura colocará as instruções juntas em um arranjo único. Caixas sombreadas representam registradores. Linhas grossas representam barramentos de 16 bits.

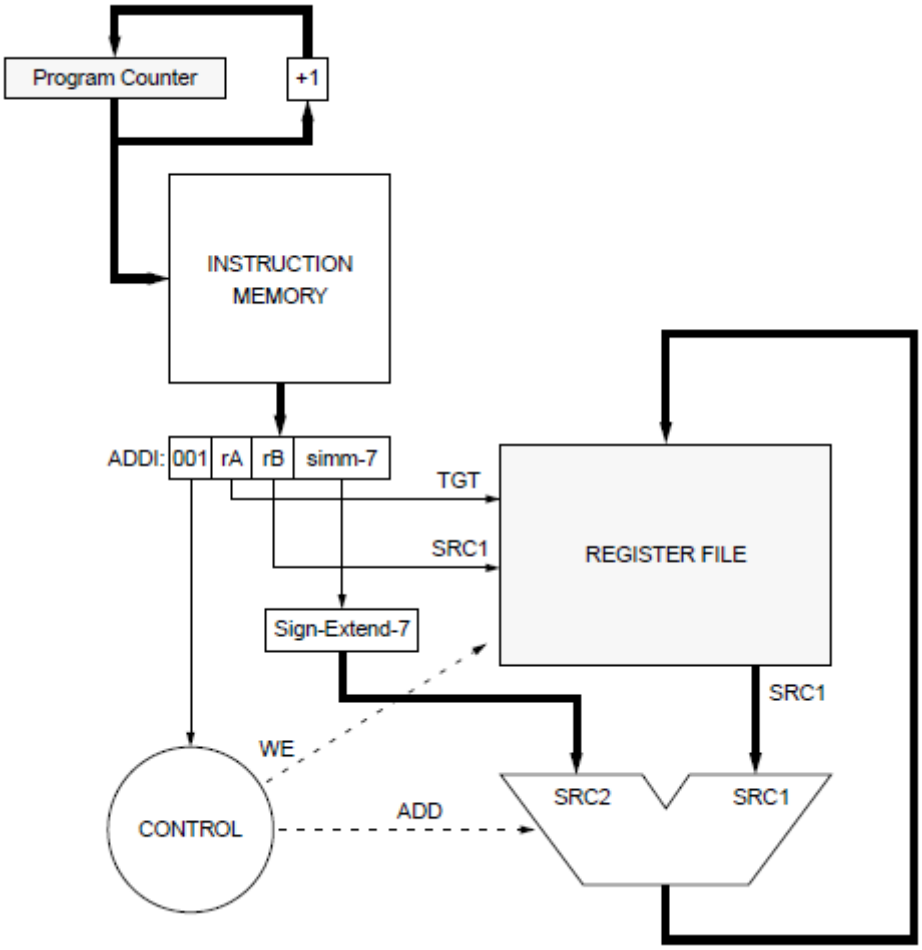
ADD



A figura ilustra o fluxo de controle para a instrução ADD. Todas as três portas do arquivo de registradores são usadas, e o bit write-enable* (WE) é ligado para o arquivo de registradores. O sinal de controle da ALU é uma simples função ADD.

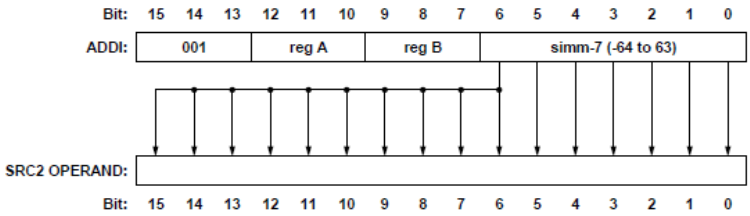
* Write-enable pode ser entendido como escrita permitida, ou seja, é permitida a escrita de conteúdo no arquivo de registradores.

ADDI

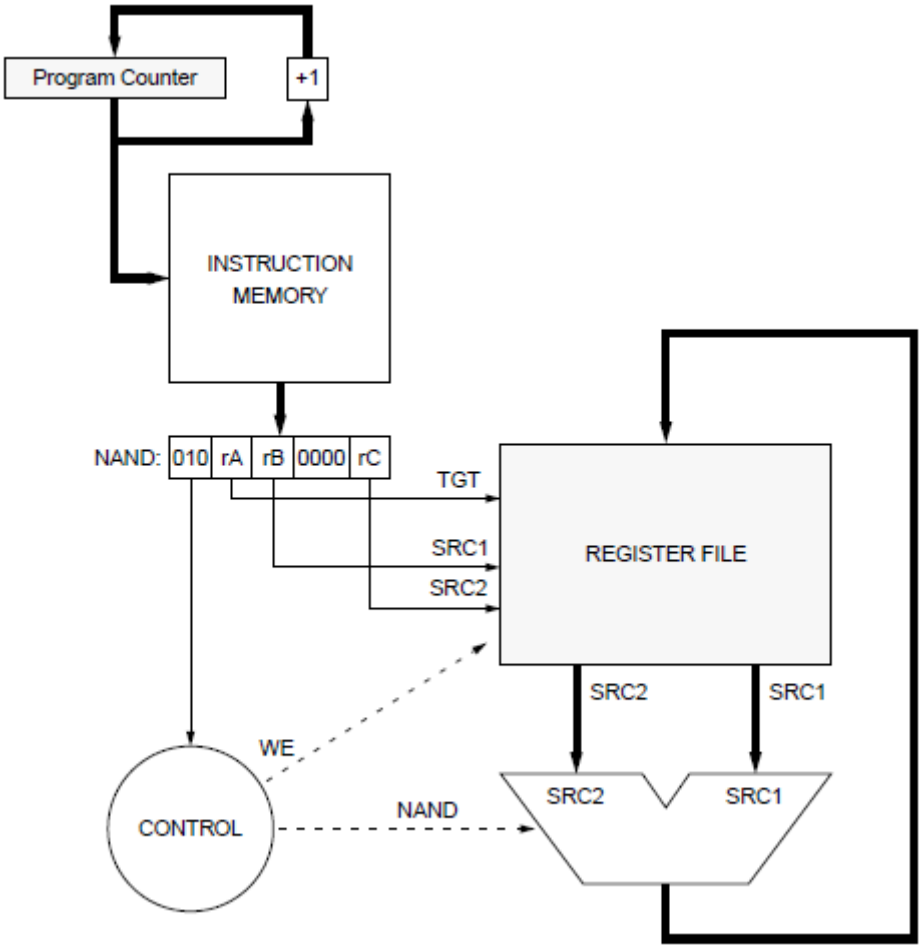


Esta figura ilustra o fluxo de controle da instrução ADD IMMEDIATE. Somente duas portas do arquivo de registradores são usadas; o segundo operando vem diretamente da instrução. O bit write-enable está ligado para o arquivo de registradores. O sinal de controle da ALU é uma simples função ADD.

O circuito lógico Sign-Extend-7 estende o sinal do valor imediato (em oposição a simples adição de zeros no começo) e, ao fazer isso, produz um numero em complemento de dois. O circuito lógico parece com isto:

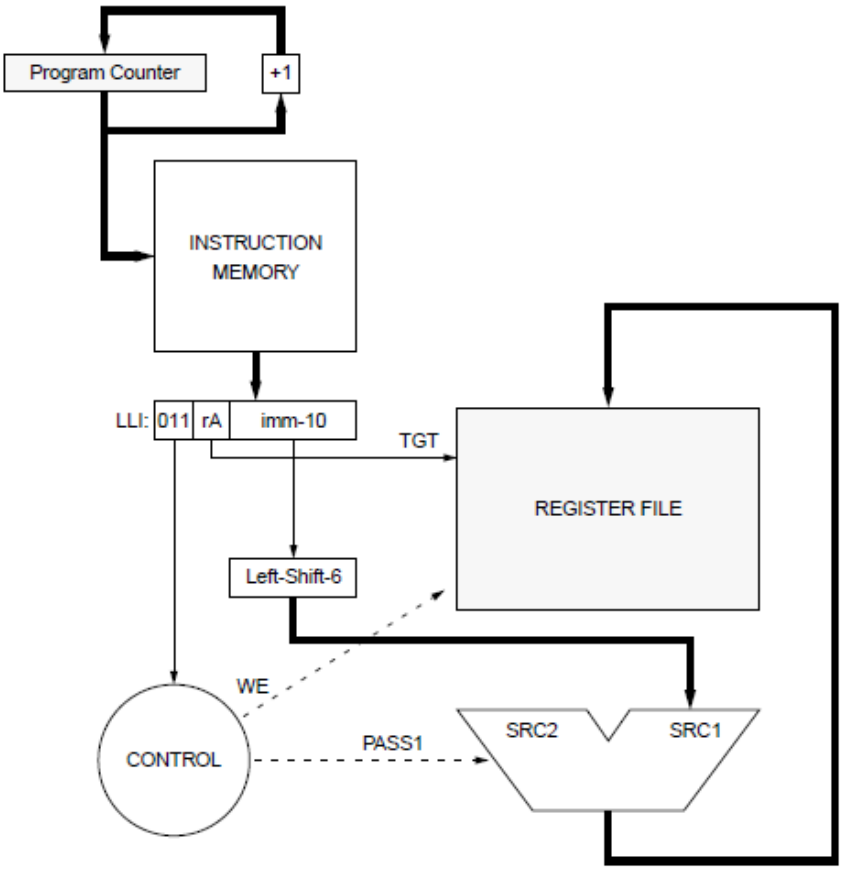


NAND



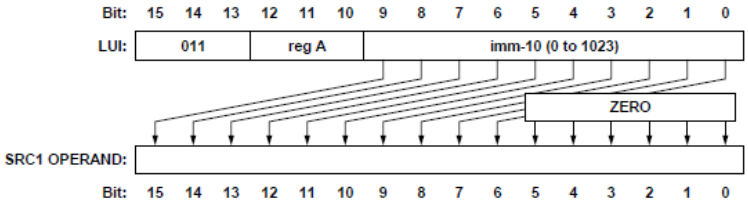
Esta figura ilustra o fluxo de controle para a instrução BITWISE NAND. Todas as portas do arquivo de registradores são usadas, e o bit write-enable está ligado. O sinal de controle da ALU é uma função BITWISE NAND.

LUI



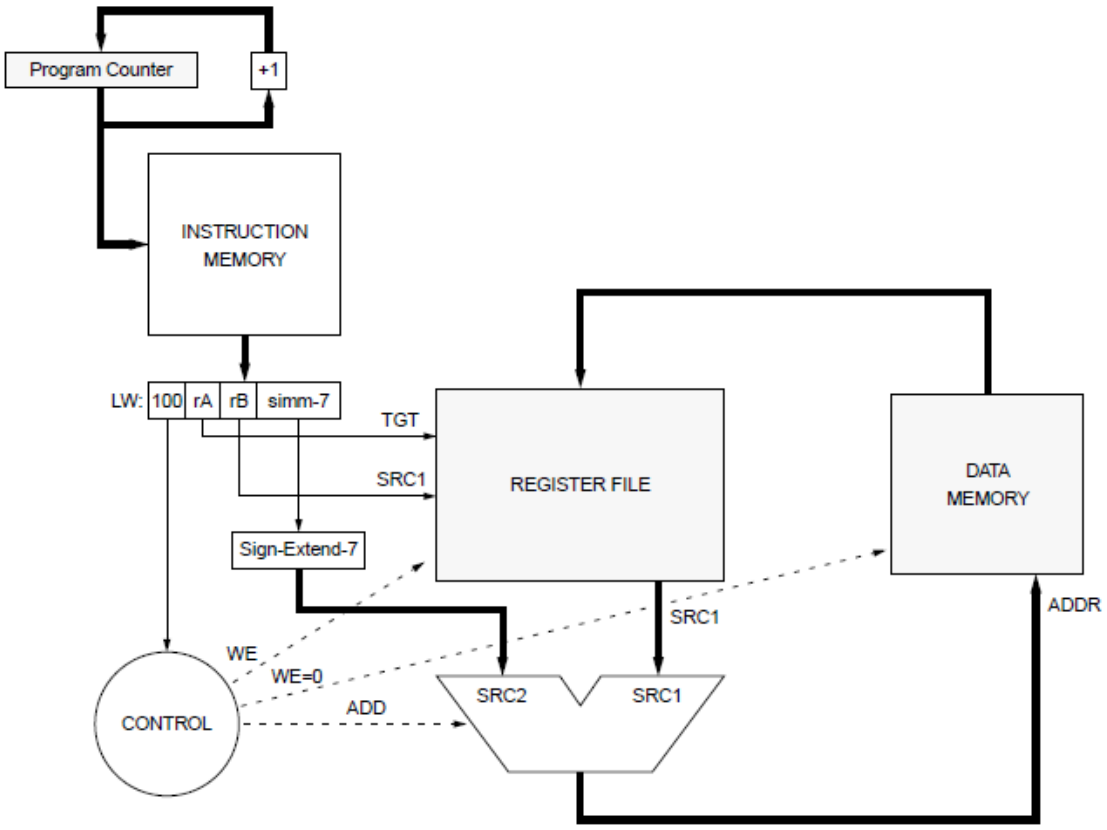
Esta figura ilustra o fluxo de controle para a instrução LOAD UPPER IMMEDIATE. Ela substitui os dez primeiros bits de uma instrução pelos 10 bits encontrados na instrução e zera os seis bits restantes. Somente duas portas do arquivo de registradores são usadas; o segundo operando é um valor imediato. O bit write-enable está ligado no arquivo de registradores.

Comparado ao ADDI, o circuito lógico de manipulação do imediato do LUI é um pouco diferente; o circuito lógico Left-Shift-6 parece com isto:



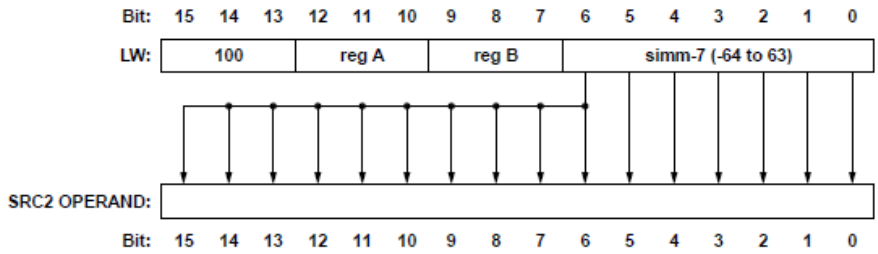
O segundo argumento da ALU (SRC2) é ignorado; isso é realizado definindo PASS1 como sendo o controle para a ALU, que envia o primeiro argumento inalterado. Este mesmo controle da ALU será usando na instrução JARL. Alternativamente poderíamos estender o valor imediato com zero e usar uma entrada deslocada na ALU enquanto entrando o valor ‘6’ na outra entrada da ALU. Contudo, o mecanismo ilustrado parece ser ligeiramente menos complicado.

LW

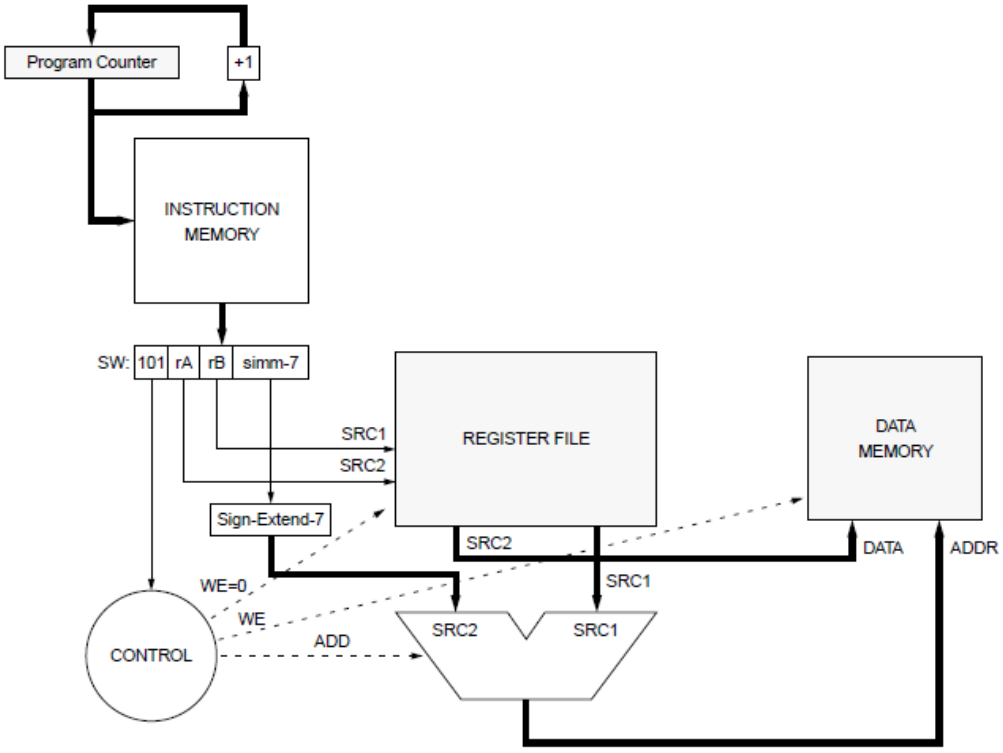


Esta figura ilustra o fluxo de controle para a instrução LOAD DATA WORD. Somente duas portas do arquivo de registradores são usadas: o segundo operando vem diretamente da instrução. O bit write-enable está ligado para o arquivo de registradores mas não para a memória de dados. O sinal de controle da ALU é simplesmente uma função ADD. O resultado da ALU ADD não vai para o arquivo de registradores, mas para a porta ADDR da memória de dados, que responde lendo o dado correspondente. Este valor está armazenado no arquivo de registradores.

O circuito lógico Sign-Extend-7 estende o sinal do valor imediato (em oposição à simplesmente adicionar zeros ao início) e produzindo um número em complemento de dois. O circuito parece com isto:



SW

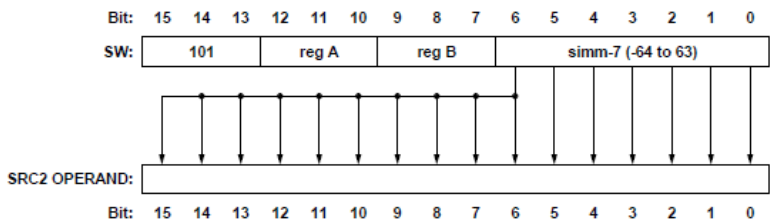


A figura ilustra o fluxo de controle para a instrução STORE DATA WORD. Somente duas portas do arquivo de registradores são usadas: a porta de escrita do arquivo de registradores (TGT) não é usada. A porta de saída SRC2 não alimenta a ALU (como faz normalmente), mas sim a entrada DATA da memória de dados. Este é o valor a ser escrito na memória. A segunda entrada da ALU é um valor imediato que vem diretamente da instrução.

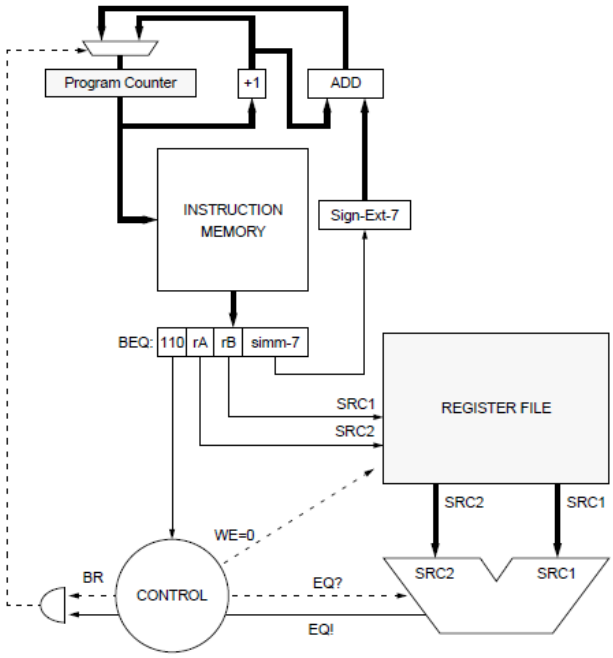
O bit write-enable está ligado para a memória de dados mas não para o arquivo de registradores. O sinal de controle da ALU é uma simples função ADD. O resultado de ALU ADD vai para a porta ADDR da memória de dados, que responde lançando o dado no endereço especificado.

Note que o campo rA da instrução, que normalmente esta ligado ao especificador TGT do arquivo de registradores, está atado ao especificador SRC2.

Como em LW, o circuito Sign-Extend-7 estende o sinal do valor imediato (ao invés de simplesmente adicionar zeros ao inicio) e produz um número em complemento de dois.



BEQ



A figura ilustra o fluxo de controle da instrução BRANCH-IF-EQUAL. Como a instrução STORE DATA WORD, ela não escreve o resultado no arquivo de registradores. Dois valores são lidos do arquivo de registradores e comparados. O sinal de controle da ALU é uma requisição para um teste de igualdade (marcado como EQ? no diagrama). Se a ALU não suporta o teste de igualdade, um sinal de SUBTRACT pode ser usado, já que a maioria das ALUs têm um sinal de saída de 1 bit que indica um resultado zero: os dois combinados produzem o mesmo resultado que um sinal EQ.

O resultado de comparação determina qual dos dois valores deve ser colocado no contador de programa. Se os dois valores no arquivo de registradores são iguais, o valor a ser lançado no contador de programas é

```
PC + 1 + (valor imediato com sinal estendido)
```

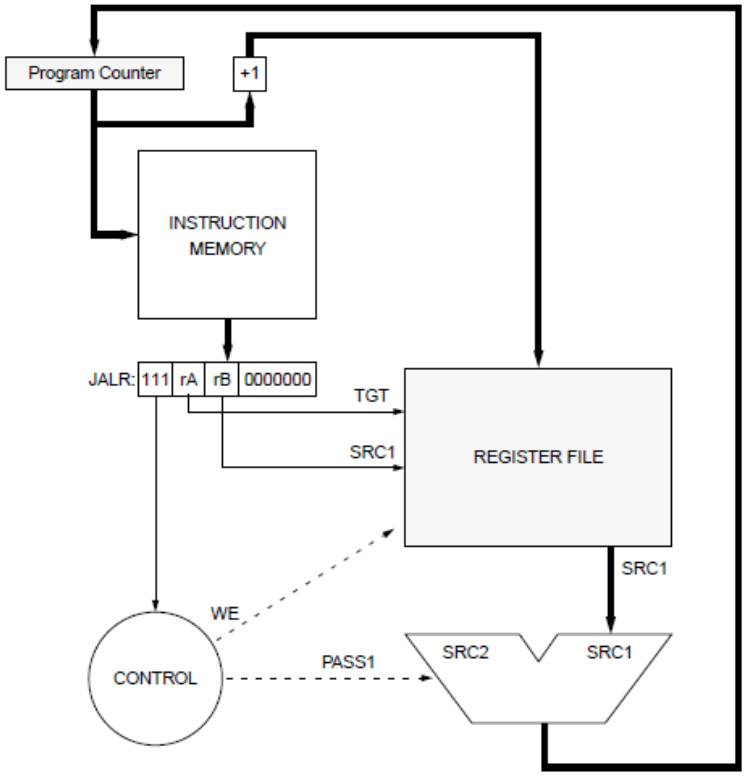
(note que a extensão de sinal é como em LW e SW). Se os dois valores lidos do arquivo de registradores não são iguais, o valor a ser lançado no contador de programa é a soma

```
PC + 1
```

(que é o valor normal de atualização do contador de programa). Nós mostramos uma pequena porção do modulo CONTROL: a porção controlando o multiplexador PC. A escolha entre PC + 1 e PC + 1 + IMM é representado por um AND da saída de EQ! da ALU e um sinal que representa CONDITIONAL BRANCH (ou seja, o opcode de 3 bits é BEQ:110). Assim, se o opcode é um desvio condicional e os dois valores são iguais, escolhe PC + 1 + IMM, caso contrário escolhe PC + 1.

Note que o campo rA da instrução, o qual está atado normalmente ao especificador TGT do arquivo de registradores está apontando para o especificador SRC2.

JARL



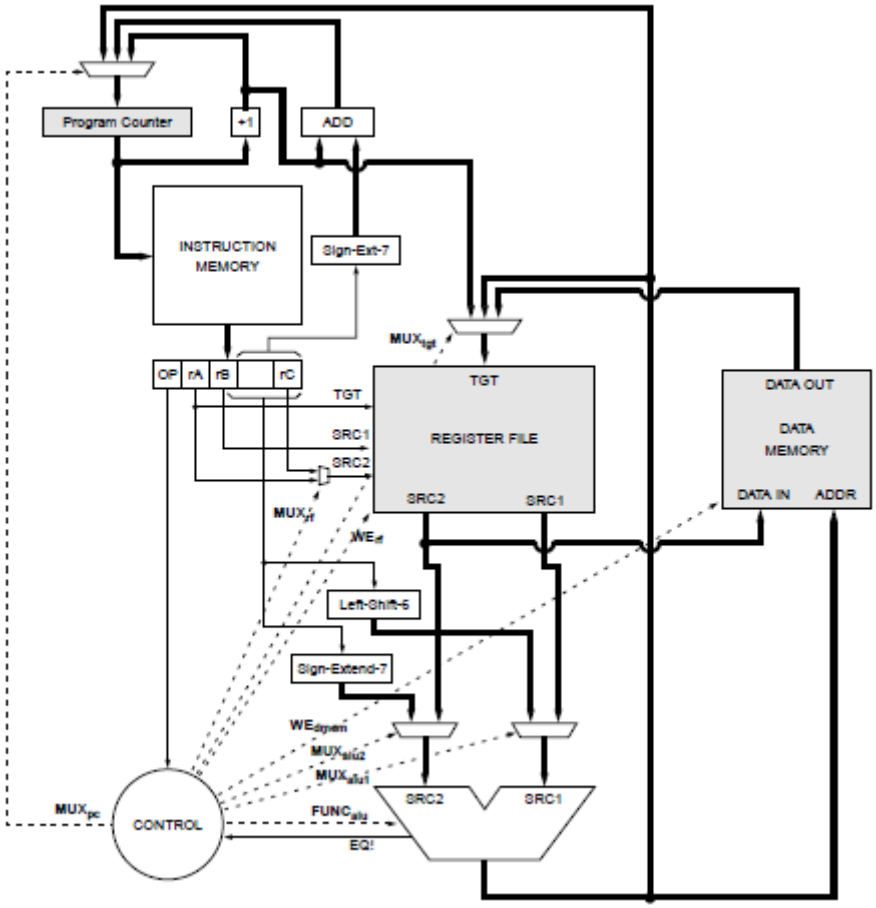
Esta figura ilustra o fluxo de controle da instrução JUMP-AND-LINK-THROUGH-REGISTER. Esta função usa duas das três portas do arquivo de registradores. Um valor é lido do arquivo de registradores e colocado diretamente no contador de programa, e a soma

```
PC + 1
```

(a qual é normalmente lançada para o contador de programas) é escrita é um registrador específico no arquivo de registradores.

O sinal da ALU é um sinal PASS assinalado para o operando SRC1 – a ALU é instruída a não executar nenhuma função, somente passar o operando SRC1 diretamente. Este valor é escrito no contador de programas.

Colocando tudo junto



O diagrama combina todos os caminhos de dados e de controle anteriores em um arranjo completo. Para acomodar todos os controles e caminhos de dados, há uma série de multiplexadores que selecionam a direção do fluxo de dados. Esses multiplexadores são operados pelo módulo CONTROL, que depende somente de duas entradas:

1. O OPCODE de 3 bits da instrução;
2. O sinal EQ! de 1 bit da ALU (a qual, para a instrução BEQ, indica se os dois operandos são iguais – para todas as outras instruções esse sinal é ignorado).

O módulo CONTROL é essencialmente um decodificador. Ele pega esses sinas de entrada e define uma série de sinais de saída que controla a ALU, uma séria de multiplexadores, e a função de escrita do arquivo de registadores e da memória de dados. No começo de cada ciclo, um novo contador de programa é travado, o que faz com que uma nova palavra de instrução seja lida e um novo opcode seja enviado ao módulo CONTROL. O módulo CONTROL simplesmente liga as linhas de controle apropriadas para a instrução, e todos os caminhos de dados e de controle são estabelecidos: depois de atrasos no arquivo de registradores, da ALU, da memória de dados e dos correspondentes multiplexadores, todos os sinais estabilizam. Neste ponto, os valores recém-criados são travados (no arquivo de registradores, na memória de

dados e/ou no contador de programa), e o novo contador de programa faz com que uma nova instrução seja lida da memória de instruções.

Estes são os sinais que o módulo CONTROL exporta:

FUNC$_{alu}$ Este sinal instrui à ALU a executar uma dada função.

MUX$_{alu1}$ Este sinal de 1 bit controla o multiplexador conectado a entrada SRC1 da ALU. O multiplexador escolhe entre a saída SRC1 do arquivo de registradores e o valor imediato deslocado a esquerda (para ser usado pela instrução LUI).

MUX$_{alu2}$ Este sinal de 1 bit controla o multiplexador conectado a entrada SRC2 da ALU. O multiplexador escolhe entre a saída SRC2 do arquivo de registrador e o valor imediato de sinal estendido (para ser usado pelas instruções ADDI, LW e SW).

MUX$_{pc}$ Este sinal de 2 bits controla o multiplexador conectado ao contador de programa. O multiplexador escolhe entre a saída da ALU ou a saída do somador + 1 que produz a soma PC + 1 em todo ciclo.

MUX$_{rf}$ Este sinal de 1 bit controla o multiplexador conectado ao especificador de operando SCR2 do arquivo de registradores, um sinal de 3 bits que determina qual dos registradores serão lidos na porta de saída de 16 bits SRC2. O multiplexador escolhe entre os campos rA e rC da instrução.

MUX$_{tgt}$ Este sinal de 2 bits controla o multiplexador conectado a porta de entrada de dados TGT do arquivo de registradores, que carrega a informação a ser escrita no arquivo de registradores (contanto que o bit write-enable do arquivo de registradores esteja ligado). O multiplexador escolhe entre a saída da ALU, a saída da memória de dados e a saída do somador +1 atado ao contador de programas.

WE$_{rf}$ Este sinal de 1 bit habilita ou desabilita a porta de escrita do arquivo de registradores. Se o sinal está alto, o arquivo de registradores pode escrever um resultado. Se ele está baixo, a escrita é bloqueada.

WE$_{dmem}$ Este sinal de 1 bit habilita ou desabilita a porta de escrita da memória de dados. Se o sinal está alto, a memória de dados pode escrever um resultado. Se ele está baixo, a escrita é bloqueada.[i]

---

[i] Todas as imagens foram retiradas do artigo *The RiSC-16 Instruction-Set Architecture* escrito pelo Prof. Bruce Jacob. O texto integral e em inglês pode ser encontrado em https://www.ece.umd.edu/~blj/RiSC/RiSC-seq.pdf