Campus | B5 7mm×30行 40页

Embedded Processor Synthesis -- ELEC6234

lk1n16@soton.ac.uk

Campus®

B5 点线本 | 7mm×30行 | 40页

KOKUYO

No.

Date

RISC architecture in embedded applications

RISC (Reduced Instruction - Set Computer)

RISC is a microprocessor that is designed to perform a smaller number computer instruction so that it can operate at a higher speed.

John Cocke of IBM Research in Yorktown, New York, noticed that about 20% of the instruction in a computer did 80% of the work.

SPARC Scalable Processor Architecture
MIPS Microprocessor without Interlocked Pipeline stage
ARM Advanced RISC Machine

RISC Characteristic

① Small instruction set — the instruction set contains simple, basic instructions; more complex operations are performed using sequences of simple instructions
② Fixed length instruction — each instruction is the same length, so that it may be fetched in a single operation
③ 1 clock-cycle instruction — most instructions complete in one clock cycle. This is achieved through pipelining which allows the processor to handle several instructions at the same time. Pipeline is the key technique used in RISC machines to improve performance

Pipelining : a key RISC technique

Pipelining is a design technique where the computer's hardware processes more than one instruction at a time, and doesn't wait for one instruction to complete before starting the next.

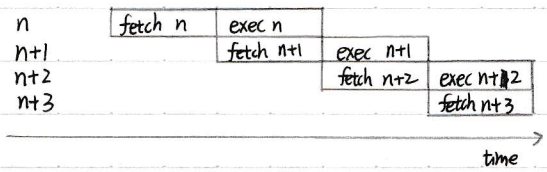
RISC machine has the same four stages of instruction execution as in our typical CISC machine: 取得 fetch, decode, execute & write. But these stages are executed in parallel. As soon as one stage completes, it passes on the result to the next stage and then begins working on another instruction.

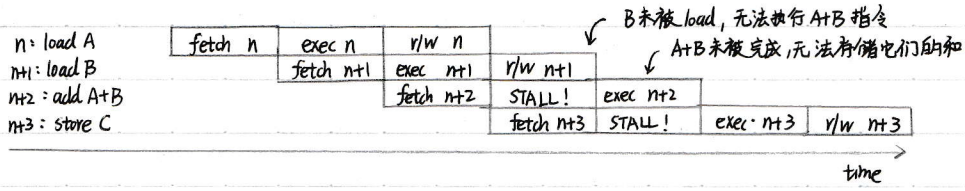
In a typical pipelined RISC design, each instruction takes 1 clock cycle for each stage, so the processor can accept 1 new instruction per clock.

Pipelining issues

Pipelining enhances performance

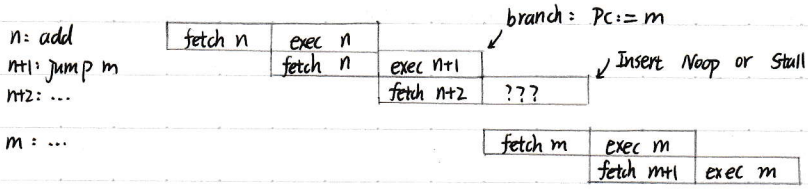
[Scenario 1] a two-stage pipeline: { Fetch : retrieve instruction from memory; Execute : perform an ALU operation with register data.





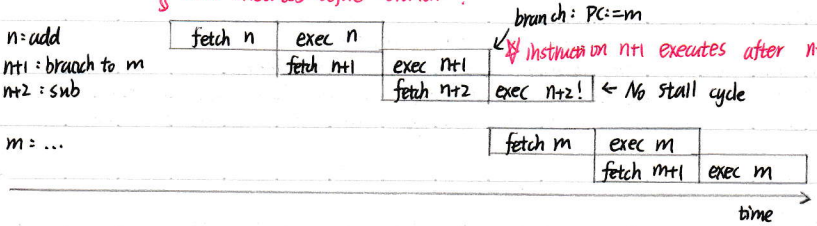
[Scenario 2] a three-stage pipline for load/store — Fetch : retrieve instruction from memory; Execute : calculate memory address; Read/Write : transfer data from/to memory to/from CPU

[Branching and pipelines] what to do with pre-fetched instructions if there is a branch?



[Delayed branch] pipeline optimisation: execute the instruction after the branch to increase pipeline efficiency.

↳ 'sub' executes before 'branch'!



Main RISC advantages for embedded applications

[Simpler hardware] - As the instruction set of a RISC processor is so simple, it uses up much less chip space; extra functions, such as memory management units or floating point arithmetic units, can also be placed on the same chip cost dramatically
- Smaller chips allow a semiconductor manufacturer to place more parts on a silicon wafer, which can lower the per-chip

[Shorter design cycle] Since RISC processors are simpler than equivalent CISC processors, they can be designed more quickly, and can take advantage of other technological developments sooner than corresponding CISC designs, leading to greater leaps in performance between generations

[Low energy consumption] Energy has become the primary performance characteristic in embedded designs, especially those aimed at mobile consumer markets. Here RISC processors have a clear advantage over CISC designs: simpler hardware, smaller control logic are features that naturally lead to low power consumption.

RISC's disadvantages

[Code quality] - The performance of a RISC processor depends greatly on the quality of the code that is executing.
- RISC processors are harder to program efficiently than their CISC equivalents. If the instruction scheduling in a program is poor, the processors can spend quite a bit of time **stalling**: wait for the result of one instruction before it can proceed with a subsequent instruction
- Instruction scheduling rules can be complicated in RISC processors, hence the performance of a RISC application depends critically on the quality of the code generated by the **compiler**.

[System Design] - They require more instructions, and hence memory, than CISCs to implement applications.
- RISC processors require very fast memory systems to feed them instructions. RISC-based systems typically contain large memory caches, usually on the chip itself. This is known as a **first-level cache**.

No.

Date

Why is CISC still around ?

Why are these still CISC CPUs being developed ?
Why is intel spending time and money to manufacture new version of the architecture which started in 1970s ?
Answer : buckward compatibility (向后兼容)
The IBM compatible PC is the most common computer in the world. Intel wanted a CPU that would be run all applications that are in the hands of more than 100 million users.

CISC - RISC convergence

Much of the RISC philosophy has been adopted in recent CISC architecture : e.g. pipelines, branch prediction, hardware stacks.
Recent RISC developments, such as high degree of pipelining and parallelism, do not necessarily lead to simple control structures.
Modern RISCs and CISCs do not tend to have vastly different clock speeds.
The real performance issue seems to lie not in architecture but in optimization of instruction sets and details of machine organization

Today, most CISC processors are based on hybird CISC-RISC architecture
These designs use a decoder to convert CISC instructions into RISC instructions before execution. They are then processed by a RISC core, which performs a few basic instructions very quickly.
Having a RISC core is advantageous because it allows performance enhancing features, such as pipelining and branch prediction.

ARM

MIPS

SPARC

PowerPC

Ultra-low energy bit-serial architectures

is an instruction set architecture for a processor that has datapath widths and data register widths of 1 bit wide

#1-bit (bit-serial) processors

Were used between 1940s and 1960s when hardware was very expensive. More recently they have not been considered seriously because they are slow.

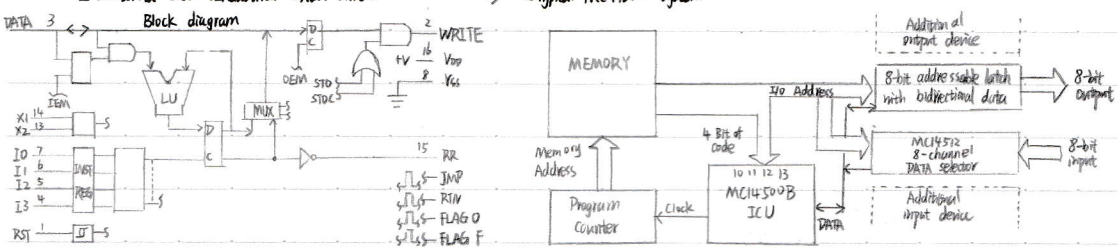
However, the have several advantages that may be useful in many-core context

@ Small physical size

@ Less power, also overall energy for computation might be less, but needs to be investigated further.

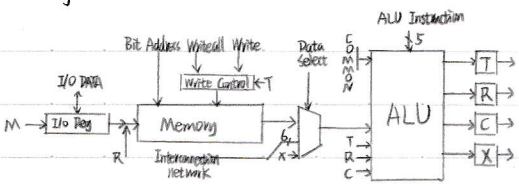
#1-bit processors have been used in CMOS era

△ Motorola ICU (Industrial Control Unit) ⇒ △ Typical MC 145000 system



#Recent bit-serial machines

Processing element (PE)



#Advantages of single-bit processing

Extreme simplicity

Very low power consumption

Many-core 1-bit systems can be reconfigured easily

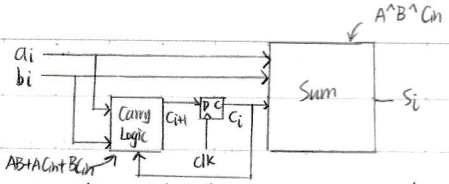
- Vector processing (Single Instruction - Multiple Data path) is straightforward
- Pipelining for higher performance is straightforward
- Networking using single-bit data transfer (eg. SPI) is straightforward
- No huge number of 'wires' between ~~cores~~ cores (No huge energy losses in wiring)
- No complex switching for reconfiguration (Further space and energy savings)

No.

Date

1-bit adder gate-level diagram

$$\begin{cases} Sum = A \oplus B \oplus Cin \\ Cout = AB + ACin + BCin \end{cases}$$



Single-bit serial adder for multi-bit addition

Serial multiplication

3×5=15

```
  011
 x101
  011
 000
+011
 01111
```

$a_2\ a_1\ a_0$ × $b_2\ b_1\ b_0$

$a_2b_0\ \ a_1b_0\ \ a_0b_0$

$a_2b_1\ \ a_1b_1\ \ a_0b_1$

$a_2b_2\ \ a_1b_2\ \ a_0b_2$

$m_1\ \ m_0$

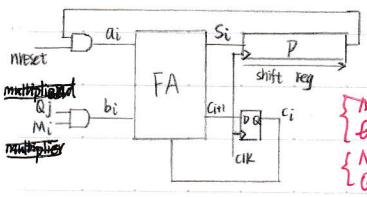
Algorithm for $m = A \times B$:

① Set $m^{(0)} = 0$

② for $m^{(i+1)}$: perform $m^{(i+1)} = m^i + Ab_i \cdot 2^i$

```
m0 = 00000
     00000
m1 = 00011 = 00011
     00011
m2 = 00000 = 00011
     00011
m3 = 01100 = 01111
```

1-bit multiplication



Uses n^2 clock cycles

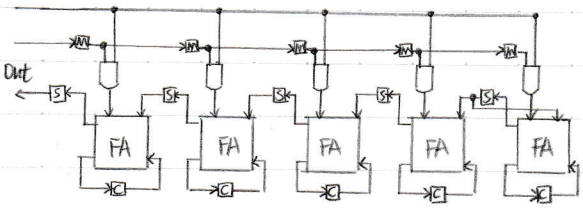
Steps for M × Q:

1. Clear product P (nreset=0, Q_i=0, for n cycles)
2. for j=0 to n-1 ; // $a_i = M_i$, if Q_j=1 otherwise no add, shift P only
   for i=0 to n-1 :
   clock carry register and Product register

~~M: Multiplier~~
~~Q: Multiplicant~~
M: Multiplicant
Q: Multiplier

Enhanced bit-serial multiplier

In a parallel bit-serial processor, several PEs can be used to ~~execute~~ accelerate multiplication



[Bit-serial processing] always shift after finishing one bit of Qj

i	j	Mi	Qj	C	PH	PL
0	0	1	1	0	000	000
0	0	1	1	0	001	000
1	0	1	1	0	011	000
2	0	1	1	0	111	000
					011	100
0	1	1	0	0	011	100
1	1	1	0	0	011	100
2	1	1	0	0	011	100
					001	110
0	2	1	1	0	000	110
1	2	1	1	1	000	110
2	2	1	1	1	000	110
					100	011

加完右移 (个位加完"+"十位)

Mi & Qj

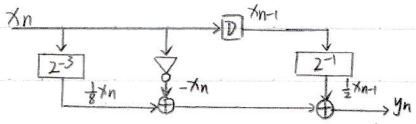
Bit-serial filter example

Bit-serial architectures are used nowadays for addition and multiplication in DSP e.g. digital filters

Example: pipelined bit-serial FIR filter: $y(n) = -\frac{7}{8}x(n) + \frac{1}{2}x(n-1)$

constant coefficient multiplications are implemented as shifts and adds

$y(n) = -x(n) + x(n)\cdot 2^{-3} + x(n-1)\cdot 2^{-1}$



Shift and add parallel multiplication

111 M	
X101 Q	
111000	ADD
011100	Shift
001110	Shift
100110	ADD
100011	Shift

按Q的位 { 0: 只shift / 1: 加上M然后shift

因为Q是按位判断，用一位扔一位，而shift也是逐次判断结束后移位，所以可以直接利用Q的无用位做移位

※ Double length accumulator AQ shares storage with multiplier Q

Assignment Module a neurons

weights will be given

Carry-out vs Overflow ← Unsigned number Carryout ≠ Overflow

No.

Date

ARM Embedded Processor Cores

Data Sizes and Instruction Sets

ARM is a 32-bit load/store RISC architecture
- The only memory accesses allowed are loads and stores
- Most internal registers are 32 bits wide.
- Most instructions execute in a single cycle.

When used in relation to ARM cores
- Halfword means 16 bits (two bytes)
- Word means 32 bits (four bytes)
- Doubleword means 64 bits (eight bytes)

ARM cores implement two basic instruction sets
- ARM instruction set - instructions are all 32 bits long
- Thumb instruction set - instructions are a mix of 16 and 32 bits
  Thumb-2 technology added many extra 32- and 16-bit instructions to the original 16-bit Thumb instruction set.

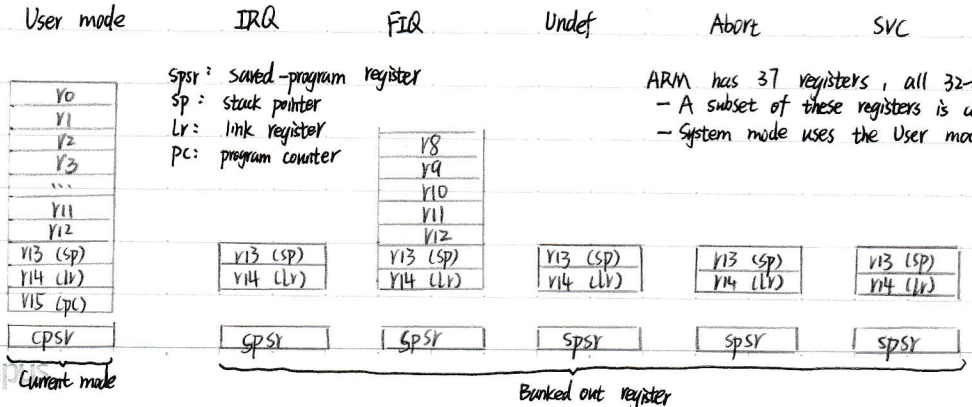
Depending on the core, many also implement other instruction sets
- VFP instruction set - 32 bit (vector) floating point instructions
- NEON instruction set - 32 bit SIMD instructions
- Jazelle-DBX - provides acceleration for Java VMs (with additional software support)
- Jazelle-RCT - provides support for interpreted languages.

Processor Modes

ARM has seven basic operating modes
- Each mode has access to its own stack space and a different subset of registers
- Some operations can only be carried out in a privileged mode

	Mode	Description	
Exception Modes	Supervisor (SVC)	Entered on reset and when a Supervisor call instruction (SVC) is executed	Privileged Modes
	FIQ	Entered when a high priority (fast) interrupt is raised	
	IRQ	Entered when a normal priority interrupt is raised.	
	Abort	Used to handle memory access violations	
	Undef	Used to handle undefined instructions	
	System	Privileged mode using the same registers as User mode.	
	User	Mode under which most Applications/OS tasks run	Unprivileged Mode

The ARM Register Set



SPSR: Saved-program register
SP: stack pointer
LR: link register
PC: program counter

ARM has 37 registers, all 32-bits long
- A subset of these registers is accessible in each mode
- System mode uses the User mode register mode

字节顺序，端序，尾序

Endianness: refers to the sequential order used to numerically interpret a range of bytes in computer memory as a larger, composed word value.

[APSR: Application Program Status Register]

APSR holds copies of the Arithmetic Logic Unit (ALU) status flag. They are also known as the condition code flags. They are used to determine whether conditional instructions are executed or not.

[Current Program Status Register]

CPSR holds ① the APSR flags ② the current processor mode ③ interrupt disable flags ④ current processor state ⑤ endianness state ⑥ execution state bits for the IT block.

Only the APSR flags are accessible in all modes.

The endianness bit (E) of the CPSR is accessible only in privileged software execution. It can be read by MRS and written by MSR, but SETEND is the preferred instruction to write to the E bit.

[SPSRs: Saved Program Status Registers]

The SPSR is used to store the current value of the CPSR when an exception is taken so that it can be restored after handling the exception. Each exception handling mode can access its own SPSR. User mode and System mode do not have an SPSR because they are not exception handling modes.

[SP, the stack pointer]

Register R13 is used as pointer to the active stack.

In Thumb code, most instructions cannot access SP. The only instructions that can access SP are those designed to use SP as a stack pointer. The use of SP for any purpose other than as a stack pointer is likely to break the requirements of operating systems, debuggers, and other software systems, causing them to malfunction.

[LR, the Link Register]

Register R14 is used to store the return address from a subroutine. At other times, LR can be used for other purposes.

When a BL or BLX instruction performs a subroutine call, LR is set to the subroutine return address. To perform a subroutine return, copy LR back to the program counter. This is typically done in one of two ways, after entering the subroutine with BL or BLX instruction:

Return with a BX LR instruction.

On subroutine entry, store LR to the stack with an instruction of the form: PUSH {, LR} and use a matching instruction to return: POP {, PC}...

Memory saving is particularly important on a part with on-chip code memory, and no external memory interface
→Thumb code is able to provide up to 65% of the code size of ARM and 160% of the performance of an equivalent ARM processor connected to a 16-bit memory system.

[The Thumb instruction set]

The thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. Thumb instructions are each 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model. Thumb instructions operate with the standard ARM register configuration, allowing excellent interoperability between ARM and Thumb states.

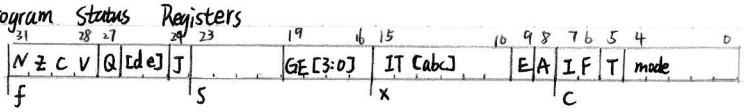
On execution, 16-bit Thumb instructions are transparently decompressed to full 32-bit ARM instructions in real time, without performance loss.

Thumb has all the advantages of a 32-bit core. Therefore it offers a long branch range, powerful arithmetic operations, and a large address space.

Instruction fetching is slower than code execution. Effectively more thumb mode instructions are required compared to ARM mode, but in thumb mode code size is less and therefore more instructions can be executed in the same time.

Thumb code: the focus is code size since on-chip flash ROM is limited.

Program Status Registers



Single Instruction Multiple Data

[Condition code flags]
- N : Negative result from ALU
- Z : Zero result from ALU
- C : ALU operation Carried out
- V : ALU operation overflowed

Unsigned number ≠ carry-out ≠ overflow

[Sticky Overflow flag - Q flag]
- Indicate if saturation has occurred

[SIMD Condition code bits - GE[3:0]]
- Used by some SIMD instructions

T bit : { T=0 Processor in ARM state; T=1 Processor in Thumb state }

J bit : J=1 Processor in Jazelle state

E bit { E=0 : Data load/store is little endian; E=1 : Data load/store is bigendian }

A bit : A=1 Disable imprecise data aborts

[IF THEN status bits - IT[abcde]]
- Controls conditional execution of Thumb instructions

Mode bits : Specify the processor mode

Interrupt disable bits { I=1 Disables IRQ; F=1 Disables FIQ }

Instruction Set basics

The ARM Architecture is a Load/Store architecture
- No direct manipulation of memory contents
- Memory must be loaded into the CPU to be modified, then written back out.

Cores are either in ARM state or Thumb state.
- This bit determines which instruction set is being executed.
- An instruction must be executed to switch between states.

The architecture allows programmers and compilation tools to reduce branching through the use of conditional execution.
- Method differs between ARM and Thumb, but the principle is that most (ARM) or all (Thumb) instructions can be executed conditionally.

Data Processing Instructions

There instructions operate on the contents of registers - They DO NOT affect memory.

	Arithetic		Logical		Move
manipulation (has destination register)	ADD (ADC)	SUB (SBC, RSB, RSC)	AND (BIC)	EOR (ORR, ORN)	MOV (MVN)
comparison (set flags only)	CMN (ADDS)	CMP (SUBS)	TST (ANDS)	TEQ (EORS)	

Syntax:

<Operation>{<cond>}{S} {Rd,} Rn, Operand2

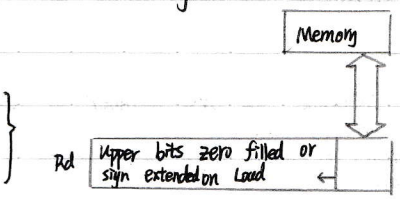
Examples:

```
- ADD r0, r1, r2    ; r0 = r1 + r2
- TEQ r0, r1        ; if r0 = r1, Z flag will be set
- MOV r0, r1        ; Copy r1 to r0
```

Single Access Data Transfer

Use to move data between one or two registers and memory.

Load Reg Double	Store Reg Double	
LDRD	STRD	Doubleword
LDR	STR	Word
LDRB	STRB	Byte
LDRH	STRH	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load



Syntax: { LDR {<size>} {<cond>} Rd, <address>
STR {<size>} {<cond>} Rd, <address>

Example: LDRB r0, [r1] ; load bottom byte of r0 from the byte of memory at address in r1

Multiple Register Data Transfer

These instructions move data between multiple registers and memory

Syntax: <LDM | STM> {<addressing_mode>} {<cond>} Rb{!}, <register list> (Rb: Base Register)

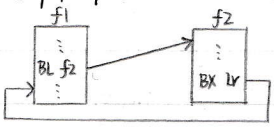
4 addressing modes → IA: Increment address After each transfer. This is the default, and can be omitted
IB: Increment address Before each transfer (ARM only)
DA: Decrement address After each transfer (ARM only)
DB: Decrement address After Before each transfer

Eg: { LDM r10, {r0, r1, r4} ; load registers, using r10 base
PUSH {r4-r6, pc} ; store registers, using sp base,

Also: Push/Pop, equivalent to STMDB/LDMIA with SP! as base register.

↑ Writeback is specified with the '!' suffix

Subroutines



Implementing a conventional subroutine call requires two steps { Store the return address; Branch to the address of the required subroutine.

These steps are carried out in one instruction, BL { The return address is stored in the link register (Lr / r14); Branch to an address (range dependent on instruction set and width.)

Return is by branching to the address in Lr

op1
- B — Branch
- BL — Branch with link
- BLX — Branch with link, and exchange instruction set

} copy the address of next instruction into Lr (r14)

op2
- BX — Branch and exchange instruction set
- BLX — Branch with link, and exchange instruction set
- BXJ — Branch, and change to Jazelle execution

} These instructions can be used as branches in Thumb-2EE code, but cannot be used to change state.

No.

Date

Supervisor Call (SVC)

SVC {<cond>} <SVC number>

Causes an SVC exception.

The SVC handler can examine the SVC number to decide what operation has been requested.
But the core ignores the SVC number
By using the SVC mechanism, an operating system can implement a set of privileged operations (system calls) which applications running in user mode can request.
Thumb version is unconditional.

Exception Handling

When an exception occurs, the core
- Copies CPSR into SPSR_<mode>
- Sets appropriate CPSR bits
  - Change to ARM state (if appropriate)
  - Change to exception mode
  - Disable interrupts (if appropriate)
- Stores the return address in LR_<mode>
- Sets PC to vector address

To return, exception handler needs to
- Restore CPSR from SPSR_<mode>
- Restore PC from LR_<mode>

...	
0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Supervisor Call
0x04	Undefined Instruction
0x00	Reset

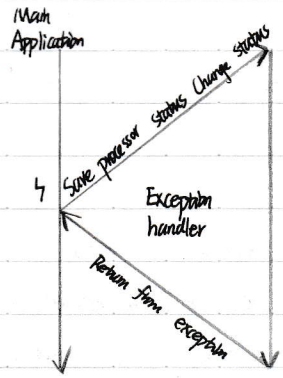
Vector Table

Cores can enter ARM state or Thumb state when taking an exception. - Controlled through settings in CP15

Exception handling process

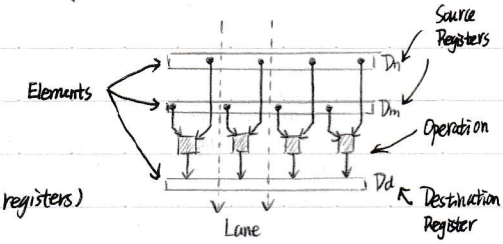
By Core
- Save processor status:
  - Copies CPSR into SPSR_<mode>
  - Stores the return address in LR_<mode>
  - Adjust LR based on exception type
- Change processor status for exception:
  - Mode field bits
  - ARM or Thumb state
  - Interrupt disable bits (if appropriate)
  - Sets PC to vector bits

By Software
- Execute exception handler: <users code>
- Return to main application:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>



What is NEON (Single Input Multiple Data)

NEON is a wide SIMD data processing architecture
- Extension of the ARM instruction set (v7-A)
- 32 x 64-bit wide registers (can also be used as 16 x 128-bit wide registers)



NEON instructions perform "Packed SIMD" processing
- Registers are considered as vector of elements of the same data type.
- Data types available: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single prec. float.
- Instructions usually perform the same operation in all lanes.

最近SIMD有点奇怪

No.

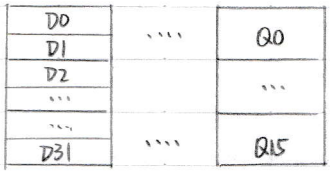
Date

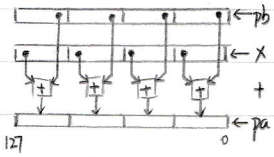
NEON Coprocessor registers

NEON has a 256-byte register file { Separated from the core registers (r0-r15) / Extension to the VFPv2 register file (VFPv3)

Two different views of the NEON registers
- 32 x 64-bit registers (D0-D31)
- 16 x 128-bit registers (Q0-Q15)

Enables register trades-offs
- Vector length can be variable
- Different registers available





向量化

NEON Vectorizing example

How does the compiler perform vectorization?

```
void add_int (int * result
        int *__restrict pa,
        int *__restrict pb,
        unsigned int n, int x)
{ unsigned int i;
  for (i=0 ; i< (n & ~3) ; i++)
    pa[i] = pb[i] + x;
}
```

n & ~3 → Make sure n ≥ 4 If NOT

① Analyze each loop
- Are pointer accesses safe for vectorization? What data types are being used?
- How do they map onto NEON vector registers?
- Number of loop iterations

② Unroll the loop to the appropriate number of iterations, and perform other transformations like pointerization.

```
void add_int (int *pa, int *pb, unsigned n, int x)
{
  unsigned int i;
  for (i=(n&~3)>>2); i; i--)
  {
    *(pa+0) = *(pb+0) + x;
    *(pa+1) = *(pb+1) + x;
    *(pa+2) = *(pb+2) + x;
    *(pa+3) = *(pb+3) + x;
    pa+=4;
    pb+=4;
  }
}
```

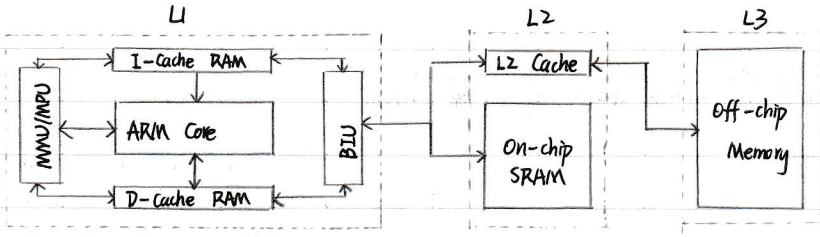
3
11

Memory Types

Each defined memory region will specify a memory type
The memory type controls the following:
- Memory access ordering rules
- Caching and buffering behaviour

There are 3 mutually exclusive memory types:
- Normal
- Device
- Strongly Ordered

Normal and Device memory allow additional attributes for specifying
- The cache policy
- Whether the region is shared
- Normal memory allows you to separately configure Inner and Outer cache policies (discussed in the Caches and TCMs module)



Typical memory system can have multiple levels of cache
- level 1 memory system typically consists of L1-cache, MMU/MPU and TCMs
- level 2 memory system (and beyond) depends on the system design.

Memory attributes determine cache behavior at different levels
- Controlled by the MMU/MPU (discussed later)
- Inner Cacheable attributes define memory access behavior in the L1 memory system
- Outer Cacheable attributes define memory access behavior in the L2 memory system (if external) and beyond (as signals on the bus)

Before caches can be used, software setup must be performed.

ARM cache features

Harvard Implementation for L1 caches — Separate Instructions and Data caches

Cache Lockdown — Prevents line Eviction from a specific Cache Way

Pseudo-random (伪随机) and Round-robin (循环) replacement strategies.
- Unused lines can be allocated before considering replacement

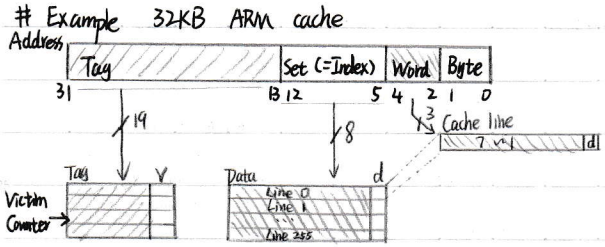
Non-blocking data cache
- Cache lookup can hit before a Linefill is complete (also checks Linefill buffer)

Streaming, Critical-Word-First
- Cache data is forwarded to the core as soon as the requested word is received in the Linefill buffer
- Any word in the cache line can be requested first using a 'WRAP' burst on the bus

ECC or parity checking

Byte = 8 bits
Word = 16 bits

Example 32KB ARM cache



v: valid bit d: dirty bit

Cache has 8 words of data in each line

Each cache line contains Dirty bit(s)
↳ Indicates whether a particular cache line was modified by the ARM core

Each cache line can be Valid or Invalid
↳ An invalid line is not considered when performing a Cache Lookup.

No.

Date

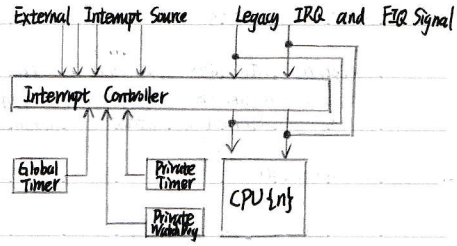
Cortex MPCore Processors

Stand Cortex cores, with additional logic to support MPCore – Available as 1-4 CPU variants

Include integrated : ① Interrupt controller ② Snoop Control Unit (SCU) ③ Timers and Watchdogs

SCU connects one to four Cortex-A9 processors to the memory system through the AXI interfaces.
- Maintain data cache coherency (相关性) between the Cortex-A9 processors
- Initiate L2 AXI memory access.
- Arbitrate between Cortex-A9 processors requesting L2 accesses.
- Manage ACP accesses

Interrupt Controller



MPCore processors include an integrated Interrupt Controller (IC)
- Implementation of the Generic Interrupt Controller (GIC) architecture

The IC provides :
- Configurable number of external interrupts (max 224)
- Interrupt prioritization and preemption
- Interrupt routing to different cores

Enable per CPU
- When not enabled, that CPU will use legacy nIRQ[n] and nFIQ[n] signals

System Timer – SysTick

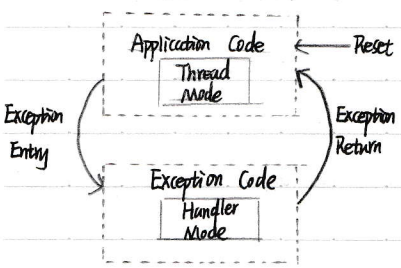
Clock source is CPU clock or optional external timing reference
- Software selectable if provided
- Reference pulse widths High/Low must exceed processor clock period
  - Counted by sampling on processor clock

Flexible System timer
- 24-bit self-reloading down counter
  - Reload on count==0
  - Optionally cause SysTick interrupt on count==0
- Reload Register
- Calibration value ← provides value required for 10ms interval
  - STCALIB inputs tied to appropriate value (System Tick Calibration Value Register)

↳ The STCALIB and STCLKEN/STCLK inputs relate to the provision of a SYSTICK (System Tick) capability. The system counter in Cortex-M0 core is clocked by the free-running clock (e.g. SCLK, FCLK), and it can count either the free-running clock itself, or it can count the cycles of an independent timing reference signal STCLKEN or STCLK (typically at a much lower frequency) if one has been provided on the SoC
To indicate which timing reference has been provided on the SoC, and the exact properties of that reference, the designer must supply reference information (usually statically tied off) on the STCALIB inputs.

Modes Overview

ARM Processor



Handler mode can also be re-entered on exception return

Instruction Set Examples

[Data Processing]

```
MOV   r2, r5                ; r2=r5
ADD   r5, #0x24             ; r5=r5+36
ADD   r2, r3, r4, LSL #2    ; r2=r3+(r4*4)
LSL   r2, #3                ; r2=r2*8
MOVT  r9, #0x1234           ; Upper halfword of r9 =#0x1234
MLA   r0, r1, r2, r3        ; r0 = (r1*r2)+r3
```

Logic shift left → LSL
MOVT: Move Top writes a 16-bit immediate value to the top half word of a register
MLA: Multiply-Accumulate

[Memory Access]

```
STRB  r2, [r10, r1]         ; Store lower byte in r2 at address {r10+r1}
LDR   r0, [r1, r2, LSL #2]  ; Load r0 with data at address {r1+r2*4}
```

Store Reg Byte → STRB
Load → LDR

[Program Flow]

```
BL    <label>
```

;PC relative branch to <label> location, and return address stored in LR (r14)

No.

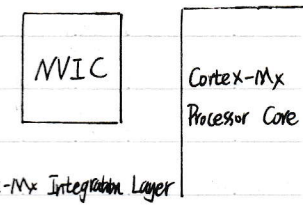
Date

Exception Handling

Exception types: ① Reset ② Non-maskable Interrupt (NMI) ③ Faults ④ PendSV ⑤ SVCall
⑥ External Interrupt ⑦ SysTick Interrupt

Exception processed in Handler mode (except Reset)
- Exceptions always run privileged.

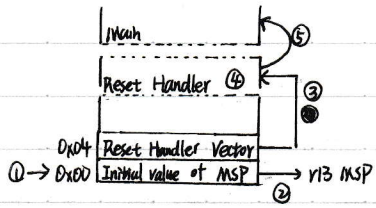
Interrupt handling
- Interrupts are a sub-class of exception
- Automatic save and restore of processor registers (xPSR, PC, LR, R12, R3-R0)
- Allows handler to be written entirely in 'C'



External Interrupts

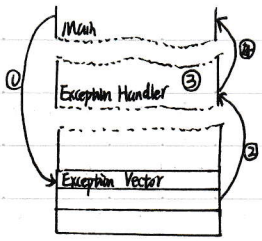
External interrupts handled by Nested Vectored Interrupt Controller (NVIC) - Tightly coupled with processor core
One Non-maskable Interrupt (NMI) supported.
Number of external interrupts is implementation-defined.

Reset Behaviour



① A reset occurs (Reset input was asserted)
② Load MSP (Main Stack Pointer) register initial value from address 0x00
③ Load reset handler vector address from address 0x04
④ Reset handler executes in Thread Mode
⑤ Optional: Reset handler branches to the main program

Exception Behaviour



① Exception occurs
- Current instruction stream stops
- Processor accesses vector table

② Vector address for the exception loaded from the vector table
③ Exception handler executes in Handler Mode
④ Exception handler returns to main.

Interrupt Service Routine Entry

Interrupt latency is the time that elapses from when an interrupt is generated to when the source of the interrupt is serviced.

When receiving an interrupt the processor will finish the current instruction for most instructions
- To minimize interrupt latency, the interrupt can take an interrupt during the execution of a multi-cycle instruction

Processor state automatically saved to the current stack.
- 8 registers are pushed: PC, R0-R3, R12, LR, xPSR
- Follows ARM Architecture Procedure Calling Standard (AAPCS) 过程调用规范

During (or after) state saving the address of the ISR is read from the Vector Table
Link Register is modified for interrupt return.
First instruction of ISR executed
- For Cortex-M3 or Cortex-M4 the total latency is normally 12 cycles, however, interrupt late-arrival and interrupt tail-chaining can improve IRQ latency.

ISR executes from Handler mode with Main stack.

Returning From Interrupt

(Same format as EXC_RETURN)

Can return from interrupt with the following instructions when the PC is loaded with 'magic' value of 0xFFFF_FFFX
- ① LDR PC,... ② LDM/POP which includes loading the PC ③ BX LR (most common)

If no interrupts are pending, foreground state is restored
- Stack and state specified by EXC_RETURN is used
- Context restore on Cortex-M3 and Cortex-M4 requires 10 cycles

If other interrupts are pending, the highest priority may be serviced
- Serviced if interrupt priority is higher than the foreground's base priority.
- Process is called Tail-Chaining as foreground state is not yet restored
- Latency for servicing new interrupt is only 6 cycles on M3/M4 (state already saved)

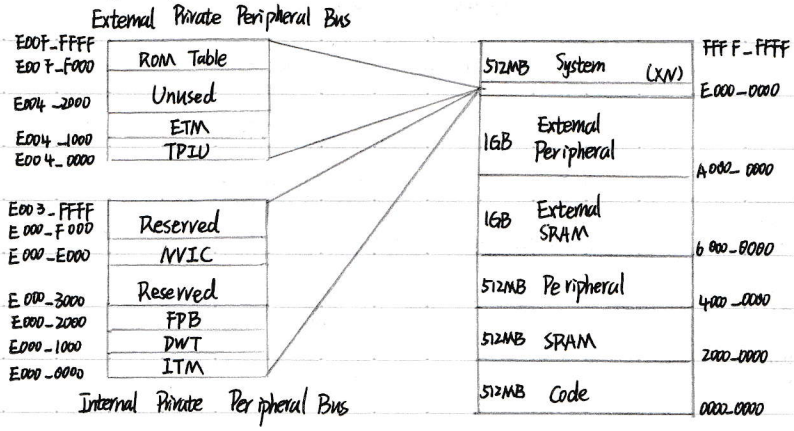
If state restore is interrupted, it is abandoned
- New ISR executed without state saving (original state still intact and valid)
- Must still fetch new vector and refill pipeline (6-cycle latency on M3/M4).

[補足]

Tail-chaining is back-to-back processing of exceptions without the overhead of state saving and restoration between interrupts. The processor skips the pop of eight registers and push of eight registers when exiting one ISR and entering another because this has no effect on the stack contents.

The processor tail-chains if a pending interrupt has higher priority than all stacked exceptions.

Processor Memory Map



Memory Types and Properties

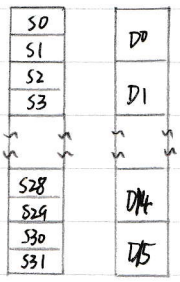
There are 3 different memory types: ① Normal ② Device ③ Strongly Ordered

Normal memory is the most flexible memory type:
- Suitable for different types of memory, for example, ROM, RAM, Flash and SDRAM
- Access may be restarted
- Caches and Write Buffers are permitted to work alongside Normal memory

Device memory is suitable for peripherals and I/O devices
- Caches are not permitted, but write buffers are still supported.
- Unaligned accesses are unpredictable (非対称読取)
- Accesses must not be restarted.

Load/store multiple instructions should not be used to access Device memory

Strongly ordered memory is similar to device memory.
- Buffers are not supported and the PPB is marked Strongly Ordered.

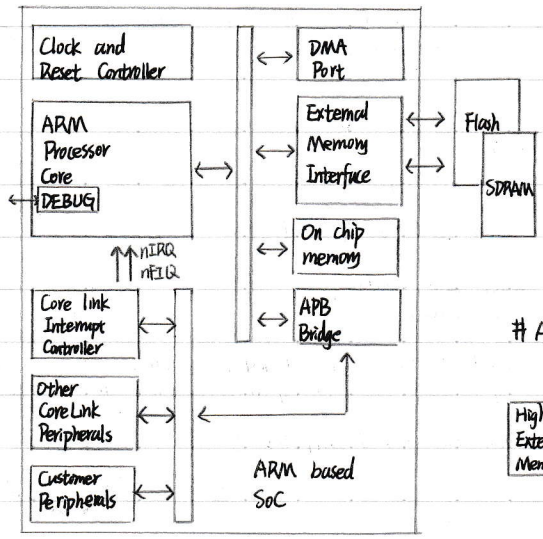


Cortex-M4F Floating Point Registers

FPU provides a further 32 single-precision registers

Can be reviewed as either { 32x32-bit registers / 16x64-bit double word registers / Any combination of the above

Example ARM-based system



ARM core deeply embedded within an SoC
- External debug and trace via JTAG or CoreSight interface

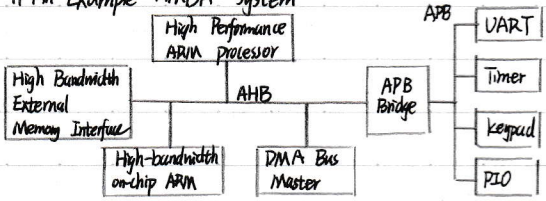
Design can be have both external and internal memories
- Varying width, speed and size - depending on system requirement

Can include ARM licensed Corelink peripherals
- Interrupt controller, since core only has two interrupt sources
- other peripherals and interfaces.

Can include on-chip memory from ARM Artisan Physical IP libraries

Elements connected using AMBA
(Advanced Microcontroller Bus Architecture)

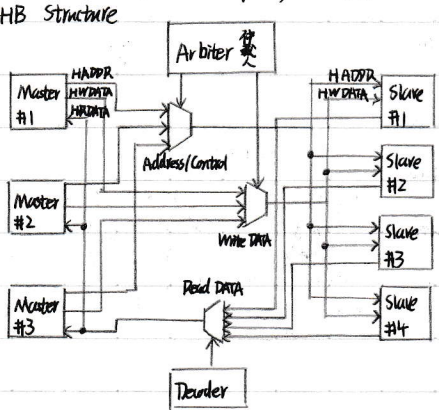
An Example AMBA System



High Performance / Pipelined
Burst Support / Multiple Bus Masters

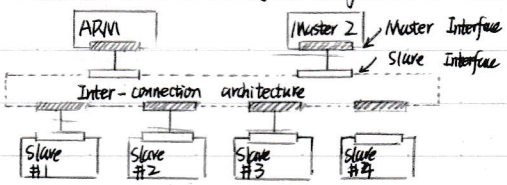
Advanced High-Performance Bus

AHB Structure



Advanced eXtensible Interface

AXI Multi-Master System Design



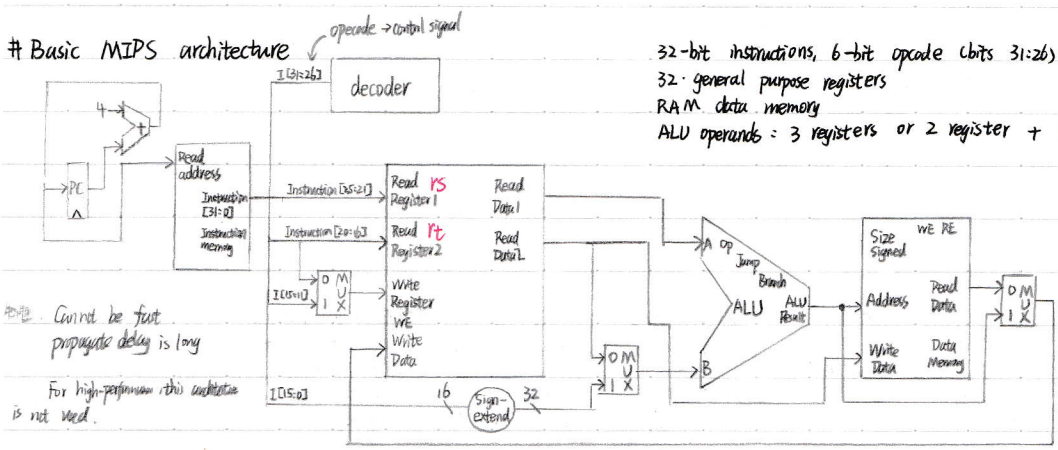
MIPS Architecture

MIPS Architecture Summary

RISC - Reduced instruction set computer, developed by MIPS Technologies in mid 1980s.
One of the simplest RISC architectures, small hardware, low power consumption
1 instruction per clock cycle in basic version.
Dual-bus architecture with 32 general purpose registers connected to ALU
Separate SRAM
A range of versions - 32-bit vs 64-bit data
- Single cycle CPU vs pipelined CPU

Recently synthesizable MIPS cores for embedded FPGA applications are being offered by a number of third party vendors

Basic MIPS architecture



32-bit instructions, 6-bit opcode (bits 31:26)
32 general purpose registers
RAM data memory
ALU operands : 3 registers or 2 register + immediate operand

MIPS Instruction formats

Type	31-bit	format	(31:0)	14:11	10:6	5:0
R	opcode (6)	rs(5)	rt (5)	rd(5)	shamt (5)	funct (6)
I	opcode (6)	rs(5)	rt(5)	immediate (16)		
J	opcode (6)	address (26)				

Instructions are divided into three types : R, I and J
Every instruction starts with a 6-bit opcode
R-type instructions specify three registers, a shift amount field, and a function field;
I-type instructions specify two registers and a 16-bit immediate value
J-type ~~instruction~~ contain a 26-bit jump target address

Sample MIPS instructions

No reg for D,C,Z,O, the value directly comes out from the ALU, branch without saving status

ADD $2, $3, $4 ; $2 = $3 + $4
- R-type ALU instruction
- Opcode is 0's, rd=2, rs=$3, rt=4, func= 000010
- 000000 00011 00100 00010 00000 000010

BEQ $3, $4, 4 ; branch if equal, i.e. $3 == $4
- I-type conditional branch instruction
- Opcode is ~~000001~~ 000100, rs=3, rt = 4, imm =4 (skip next 4 instru)
- 000100 00011 00100 0000000000000100

JALR $3 ; save PC+1 in $31, jump to [$3]
- R-type jump instruction
- Opcode is 0's, rs=3, rt=0, rd=31 (by default), func = 001001
- 000000 00011 00000 11111 00000 001001

SW $2, 128($3) ; store word
- I-type memory address instruction
- Opcode is 101011, rs= 3, rt = 2, imm = 128
- 101011 00011 00010 0000000010000000

ADDI $2, $3, 12 ; $2 = $3 + 12
- I-type ALU instruction
- Opcode is 001000, rs= 3, rt=2, imm = 12
- 001000 00011 00010 0000000000001100

J 128 ; jump to address PC ← PC_{31-28} :: 128 :: '00
- J-type pseudodirect jump instruction PC ← PC_{31-28} :: IR_{25-0} :: '00
- Opcode is 000010, 26-bit pseudodirect address is 128/4 = 32
- 000010 00000000000000000000100000

R opcode + rs + rt + rd
I opcode + rs + rt + immediate
J opcode 31:26 address 25:0

KOKUYO

No.

Date

bne : branch not equal

Synthesised instructions

Some MIPS assembler instructions don't have direct hardware implementation.

- Eg: abs $2, $3 ; $2 = abs ($3)
  Resolved to :
  - bgez $3, pos ; branch on greater or equal to 0
  - sub $2, $0, $3 ; $2 = -$3
  - j out
  - pos : add $2, $0, $3 ; $2 = $3
  - out : ...
- Eg: rol $2, $3, $4 ; $3 = $3 rotated left by $4 bits
  Resolved to :
  - addi $1, $0, 32 ; load 'b10000 to $1
  - sub $1, $1, $4 ; $1 = 'b10000 - $4
  - srlv $1, $3, $1 ; shift right right logical variable : $1 = $3 >> $1
  - sllv $2, $3, $4 ; shift left logical variable : $2 = $3 << $4
  - or $2, $2, $1 ; $2 = $2 or $1

REC 30 min

The only reason required a pipeline is to speed up the instruction execution
Not security we want in Embedded Systems
The speed depends on the propagation delay (MAXimum steps : 5)

Translation from C to MIPS machine code.

⇒ An if statement { if (i==j) f = g+h ; else f = g-h ;

[MIPS code] ; i → $1 , j → $2 , f → $3 , g → $4 , h → $5

```
    bne $1, $2, Lelse   ; if i!=j , go to Lelse
    add $3, $4, $5      ; f = g+h
    J Lexit             ; go to Lexit
Lelse : sub $3, $4, $5  # f = g-h
Lexit : ...
```

⇒ A while statement and an array : while (A[i] == c) i += 1 ;

shift left logical

[sll $d, $t, h] $d = $t << h ;

[MIPS code] ; i → $1 , c → $2 , A[] → $3
; Address of A[] in RAM → $4
; Value of A[i] → $5

```
Loop : sll $4, $1, 2     ; $4 = i*4 , prepare for byte count
       add $4, $4, $3    ; $4 = address of A[i]
       bne $1, $2, Lexit ; test while condition
       add $1, $1, 1     ; i += 1
       j Loop
Lexit: ...
```

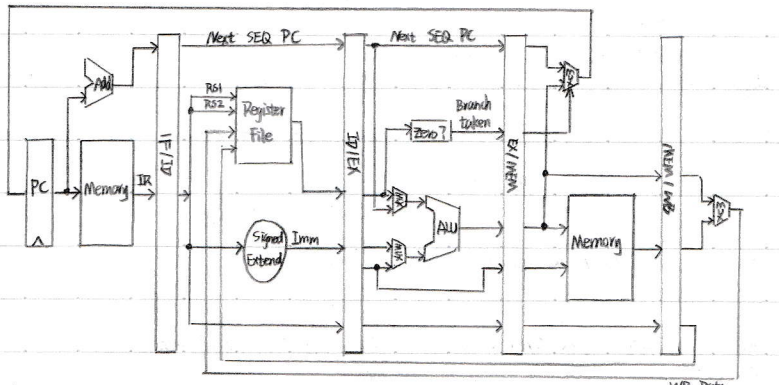
⇒ A for loop in C : for (i=0 ; i<n ; i++) A[i] = B[i] + 10 ;

[MIPS code]

```
    XOR $2, $2, $2   # $2=0 , index register (i)
    li $3, n         # $3=n , iteration limit
    sll $3, $3, 2    # multiply n by 4 (convert word count to byte count)
    li $4, a         # $4 = address of A[] (assume A[] < 2^16)
    li $5, b         # $5 = address of B[] (assume B[] < 2^16)
loop: add $6, $5, $2 # $6 = address of B[i]
    lw $7, 0($6)     # load B[i] from memory
    addi $7, $7, 10  # $7 = B[i] + 10
    add $6, $4, $2   # $6 = address of A[i]
    sw $7, 0($6)     # store $7 into A[i]
    addi $2, $2, 4   # increment i (add 4 to maintain word count)
    blt $2, $3, loop # branch if index < n (bit - branch if less than)
```

Campus

Pipelined MIPS



[MIPS pipeline stages]

- Fetch (F)
  read next instruction from memory, increment address counter
  assume 1 cycle to access memory

- Decode (D)
  read register operands, resolve instruction in control signals, compute branch target

- Execute (E)
  execute arithmetic / resolve branches

- Memory (M) – Load / Store instructions only
  Perform load/store accesses to memory
  Assume 1 cycle to access memory

- Write back (W)
  Write arithemetic results to register file.

No.

Date

(F) (D) (E) (M) (W)
Fetch → Decode → Execute → Memory → Write Back

MIPS pipeline data hazards

D & W 仅当同一个 Memory 被 Decoded 和同时

- Data hazards
  - Register values "read" in decode, writing during write-back
    - Hazard occurs when dependent instruction separated by less than 2 slots
    - Examples:
      - ADD $2, $X, $X (E)
      - ADD $X, $2, $X (D)
      - ...
      - ...

      ADD $2, $X, $X (M)
      ...
      ADD $X, $2, $X (D)
      ...

      ADD $2, $3, $4 (W)
      ...
      ...
      ADD $X, $2, $3 (D)

      Conflict
  - In most cases, data generated in same stage as data is required (EX)
    - Data forwarding
      - ADD $2 $X, $X (M)
      - ADD $X, $2, $X (E)
      - ...
      - ...

      ADD $2, $X, $X (W)
      ...
      ADD $X, $2, $X (E)
      ...

      ADD $2, $3, $4 (out-of-pipe)
      ...
      ...
      ADD $X, $2, $3 (E)

- Load hazards

  W & E

  - Stalls required when data is not produced in same stage as it is needed for a subsequent instruction
  - Examples:

    LW $2, 0 ($X) (M)
    ADD $X, $2 (E)

    被迫

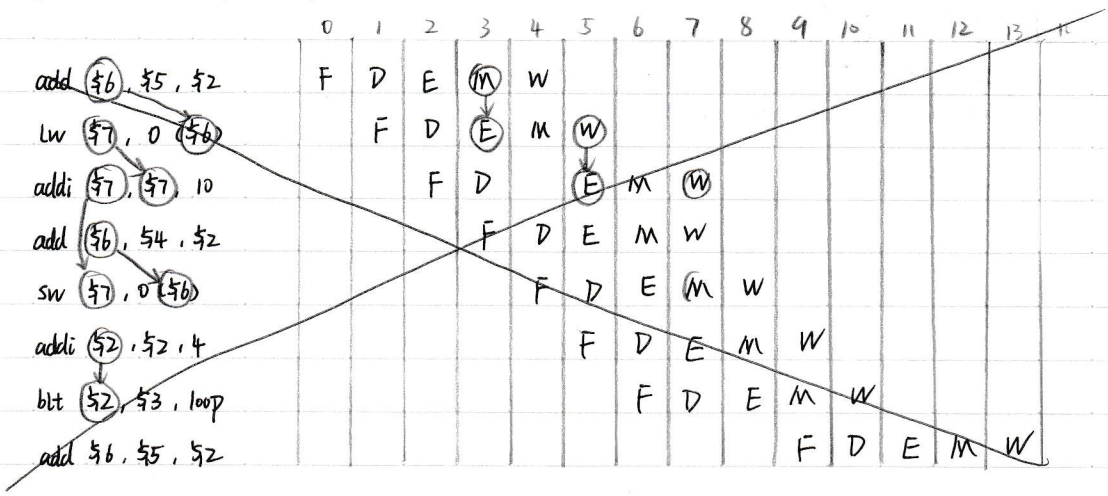
  - When this occurs, insert a "noop" into EX state, stall F and D

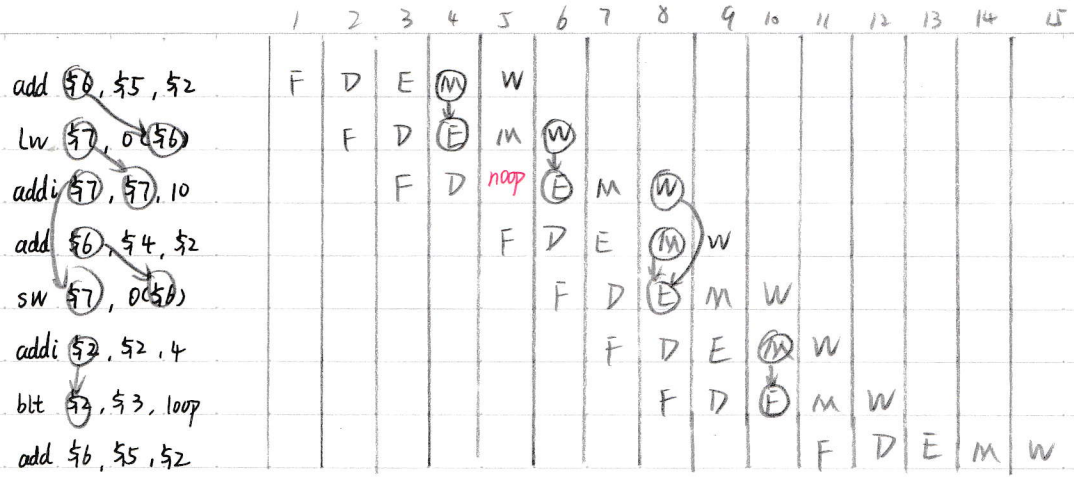
    LW $2, 0 ($X) (W)
    NOOP (M)
    ADD $X, $2 (E)

    $2未被写回，就执行加法，加法加的数是过期的数，加入一个 noop，在同一周期内，新数被上回

  - Forward from W to E

Example of MIPS pipeline operation




1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
add $0, $5, $2
lw $7, 0($6)
addi $7, $7, 10
add $6, $4, $2
sw $7, 0($6)
addi $2, $2, 4
blt $3, $3, loop
add $6, $5, $2
F D E M W
F D E M W
F D E M W
F D E M W
F D E M W
F D E M W
F D E M W
F D E M W
noop

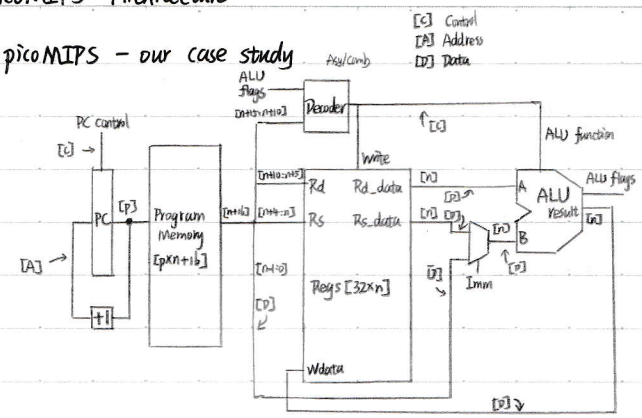
BLO : one unsigned value compares lower (LO) than another

No.

Date

PicoMIPS Architecture

#picoMIPS - our case study



Inspired by MIPS, but not a version of MIPS

Architecture suited for efficient synthesis on FPGAs with synchronous RAM blocks

User specifies the width of the data bus n, typically 8, 16, 32, 64 bits

User specifies the instruction set to be implemented in the decoder

Quiz for (i=0 ; i<=n ; i++) ;

```
.def n =0
.def i =0
LDI %i,0
LDI %n,17      Loop:
                   BLO %n,%i Loop_end   ; n≤i ?
                   ADDI %i,1
                   J   Loop
               Loop_end:
               ...
```

#picoMIPS instruction groups

ALU instructions operate on registers only, not ARM

- ADD, SUB, AND, OR, XOR, NOR
  - 2 register format : ADD %d,%s → %d = %d + %s
- ADDI, SUBI, ANDI, ORI, XORI, NORI
  - Register + Immediate : ADDI %d,%s imm → %d = %s + imm
- MUL, shift instructions SLL, SRL, SRA (will not be included in our case study)

Program flow control

- J address - unconditional absolute jump
- BEQ %d,%s address - conditional relative branch : branch if %d-%s=0
- Other conditional branches
  - BNE : branch if Z!=0
  - BGE : branch if greater than (%s > %d) BGE %s,%d = BLT %d,%s
  - BLO : branch if lower than (C̄) BHI : branch if higher C̄+Z — BHI by inverting the logic of BLO?

for unsigned

RAM related instructions

- LW %d, [address] - load word from RAM into register %d
- SW %s, [address] - store word from register %s into RAM

Other :

- E.g. NOP ; no operation
- Subroutine calls and returns, stack instructions - not implemented in our case study

General approach when synthesising on an FPGA - the decoder should only implement the instructions used by the program stored in the program memory. Hardware efficient approach

C → MIPS machine code

```
if(a==0) b=1 ; else b=2 ;
```

dedicated "zero" (some instructions)

```
⇒ .def  a=20
  .def  b=21

      BNE %a ,%0 , Lelse   ; a is register 20
      LDI %b , 1           ; b is register 21
      J Lend               ; load 1 to %b (ADDI %b,%0,1)
Lelse: ...
      LDI %b,2             ; actual code : ADDI %b,%0,2
Lend:
```

(%a: not register / constant "0"; %0: register number)

```
i=0 ; n=10 ; a=10 ;
while (i<=n) { a+= a ; i++ ;}

.def  i=16 , n=17 ,  a=18      { LDI %i,0
                                 LDI %n,10
                                 LDI %a,10

Loop:                  n≤i ?
      BLO %n ,%i , Lend     ; branch to exit loop if Carry set
      ADD %a , %a           ; a=a+a
      ADDI %i , %1          ; i=i+1
      J   Loop              ; jump to loop
Lend :
      ...
```

Steps in Embedded processors synthesis (picoMIPS case study)

Develop and test ALU with typical operations: ADD, SUB, etc.

Develop General Purpose Registers (GPR) and test

Develop a simple Instruction Decoder and test with ALU and GPR

- In the decoder implementation NOP instruction, ALU instructions, LDI

Develop program Memory

Develop program Counter

Enhance instruction Decoder with J and typical conditional branches, e.g. BEQ, BLO, etc

⇒ Basic processor core is functional at this point

Possible enhancements:

- Add data RAM
- RAM related instructions, eg. LD, ST
- Develop an I/O port, and implement in decoder IN and OUT instructions
- Add ALU hardware to support MUL instruction and shift instructions.
- Subroutine calls and returns

Your course work

- Modify the picoMIPS to implement a prescribed program
- Write a formal report
- Challenge: the smallest possible implementation on Altera Cyclone IV

FPGA Synthesis of a Microprocessor Core Notes on ALU Synthesis

Approaches to ALU Synthesis

Methods to synthesise ADD, SUB
- 1) Use a+b in 9-bits to extract carry on 9th bit
- 2) Use a+b in 8-bits, but add extra logic to extract carry from ALU MSB slice.
- 3) Synthesise ALU on structural level using 1-bit slices

For small hardware size use method 2) or 3)

Subtraction can be implemented using 2's complement subtraction rule:
- A-B = A + ~B + 1
- This way the same hardware can be used for ADD and SUB

2's complement overflow (if needed!) can be obtained from simple logic as shown in ALU code below.

Writing synthesisable SystemVerilog code for combinational logic synthesis.

When writing code for decoders or multiplexers, use conditional statements { if, then, else / case

Common trap:

Conditional statements with incompletely specified signals cannot be mapped into combinational logic:

```
if (a)
    b=1'b0;
// b is incompletely specified
```

Use the default clause:

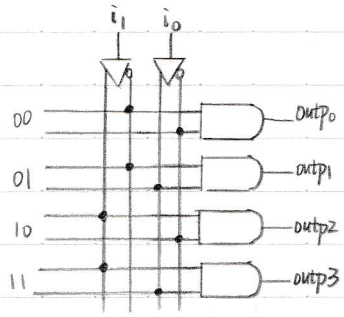
Assign default values to all signals driven by combinational logic blocks (examples will follow)

In arithmetic circuit synthesis

Arithmetic operations + - are mapped into standard adders / subtractors

Synthesis tools, e.g. Synplify Pro / Quartus map the multiplication operator * into dedicated multipliers if available, or combinational (cellular) multipliers - fast but very costly.

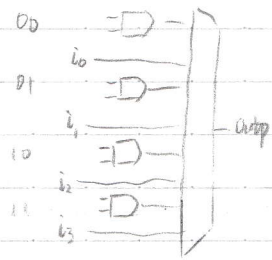
Decoder Example




```
module decoder2to4 (
    input logic [1:0] i, output logic [3:0] outp);
always_comb
begin
  outp = 4'b0000;
  unique case (i) // unique will force complete test
    0 : outp[0] = 1'b1;
    1 : outp[1] = 1'b1;
    2 : outp[2] = 1'b1;
    3 : outp[3] = 1'b1;
  endcase
end
endmodule
```

Multiplexer example

上图 outp0 ~ outp3 分别对于 $i_0 \sim i_3$ And...




```
module mux4to1 (
    input logic [3:0] i, input logic [1:0] sel, output logic outp);
always_comb
    unique case (sel)
      0: outp = i[0];
      1: outp = i[1];
      2: outp = i[2];
      3: outp = i[3];
    endcase
endmodule
```

No.

Date

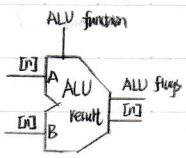
Arithmetic circuits - 8-bit adder

```
module adder8
    (input logic [7:0] A,B , input logic Cin,
     output logic [7:0] Sum , output logic Cout );

assign {Cout, Sum} = {A[7], A} + {B[7], B} + Cin ; //assign 1-extend by 1 bit
endmodule
```

Sample ALU module

```
module alu #(parameter n=8)(
    input logic [n-1:0] a,b,   //ALU operands
    input logic [2:0] func,    //ALU function
    output logic [3:0] flag,   //ALU flags V,N,Z,C
    output logic [n-1:0] result); //ALU result

always_comb
begin
  if (func == `RSUB)
     b1 = ~b + 1'b1 ;
  else
     b1 = b;
                  //Adder result
     ar = a+b1; // n-bit adder
end

always_comb
begin
//default output values ; prevent latches
     flags = 3'b0 ;
     result = a ; //default

     case (func)
      `RA : result = a;
      `RB : result = b;
      `RADD : begin:
              result = ar ;
              flags[3] = a[7]&b[7]&~result[7] | ~a[7]&~b[7]&result[7]; //V <- Overflow
              flags[0] = a[7]&b[7] | a[7]&~result[7] | b[7] &~result[7]; //C <- Carry on MSB
              end
      `RSUB : begin
              result = ar ;
              flags[3] = a[7]&~b[7]&~result[7] | ~a[7]&b[7]&result[7]; //V
              //C - note: picoMIPS inverts carry when subtracting
              flags[0] = a[7]&~b[7] | a[7]&result[7]|~b[7]&result[7] ;
              end
      `ROR : result = a|b ;
      `RAND : result = a&b ;
      `RXOR : ...
      endcase

Always-comb
      flags[1] = result == {n{1'b0}}; //Z <- Zero result
      flags[2] = result[n-1] ; //N <- Negative result

end
endmodule
```

```
`define RA   3'b000
`define RB   3'b001
`define RADD 3'b010
`define RSUB 3'b011
`define RAND 3'b100
`define ROR  3'b101
`define RXOR 3'b110
`define RNOR 3'b111
```



No.

Date

Carry Logic

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

When coding addition as A+B
We don't know Cin at MSB, but we know Sum[-] (Cin在 Synthesis中，没有单独申明)

Carry is set at MSB slice if:
A[-] & B[-] | B[-] & ~Sum[-] | A[-] & ~Sum[-] ← A或B的最高位为1，但和的最高位为0那么

In substraction A-B, complement B[-]

Detection of 2's complement overflow A[-]&B[-]&~Sum[-] | ~A[-]&~B[-]&Sum[-]

Overflow occurs in addition if the signs of both operands are the same, but the result sign is different.

No.

Date

Synthesis of registers and memory

Sequential logic synthesis

Flip-flops : Clear
Preset
ClockEnable

Registers and memory in processors
- Program Counter
- General Purpose Register File
- Instruction Register (not in picoMIPS)
- RAM - sequential RAM is modern FPGAs

RAM Synthesis

RAMs used in FPGA designs are typically arrays of latches with separate input and a separate output data bus, an address bus and a Write Enable signal; read is asynchronous.
Modern FPGA (e.g. Cyclone V) support only synchronous memory, where both read and write is synchronous.
Note that there many types of RAMs, eg.
- RAM with synchronous read
- RAM with one Enable controlling both ports
- RAM with separate Enables controlling each port
- Multiple-Port RAMs

1853 Combinational logic

Standard RAM with asynchronous read

old template

```
module ram128x8 ( input logic we,
                  input logic [6:0] address,
                  input logic [7:0] din,
                  output logic [7:0] dout );
// This 2-dimensional array defines the ram memory
logic [7:0] ram [127:0]

// write block
always_latch
  if(we)
    ram[address] <= din;
// Asynchronous read lock
assign dout = ram[address];

endmodule.
```

```
module ram128x8sync ( output logic [7:0] dout,
                      input logic [6:0] address,
                      input logic [7:0] din,
                      input logic we, clk);

logic [7:0] mem [127:0];

always_ff @ (posedge clk)
begin
  if (we)
    mem[address] <= din;

  dout <= mem [address];
end

endmodule
```

Synchronous RAM blocks in modern FPGAs

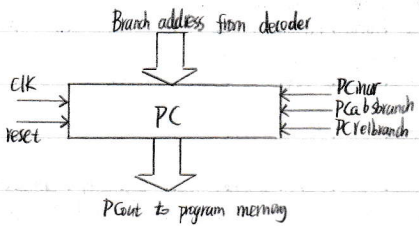
Altera Cyclone IV devices feature memory structures that consists M9K memory blocks that can be configured to provide RAM, shift registers, ROM, and FIFO buffers.
An M9K block contains 8,192 memory bits and is synchronous i.e. requires a clock.

Altera Cyclone V devices contain two types of memory blocks. Both are synchronous. configuration
① M10K blocks - 10-kilobit (Kb) blocks for larger memory
② Memory logic array (MLABs) - 640-bit memory blocks for small memories. Each MLAB can be configured as ten 32x2 blocks, giving one 32x20 simple dual-port SRAM (Static random access memory)

Synchronous RAM cannot provide the operand in one clock cycle
① Register the instruction
② Load the memory to its output

Program Counter

```
module pc #(parameter Psize=6 ) // up to 64 instructions
          (input logic clk, reset, PCincr, PCabsbranch, PCrelbranch,
           input logic [Psize-1:0] Branchaddr,
           output logic [Psize-1:0] PCount);

logic [Psize-1:0] Rbranch ; //temp variable for addition operand

always_comb //multiplexer to select next instruction offset
    if (PCincr)
      Rbranch = {{(Psize-1){1'b0}}, 1'b1}; add1
    else
      Rbranch = Branchaddr ;  //add branch addr

always_ff @ (posedge clk or posedge reset ) //async reset
    if (reset ) //reset
      PCount <= {Psize-{1'b0}};
    else if (PCincr | PCrelbranch ) // increment or branch relative
      PCount <= PCount + Rbranch;
    else if (PCabsbranch)          // absolute branch, load branch addr.
      PCount <= Branchaddr;

endmodule.
```



Program Memory

address width ↓ instruction width ↓

```
module prog #(parameter psize=6, isize=24)
            (input logic [psize-1:0] address,
             output logic [isize:0] instr );

logic [isize:0] progMem [(1<<psize)-1:0];

//Get memory contents from file
initial
   $readmemh ("prog.hex", progMem);

//Program memory read
   assign instr = progMem [address];

endmodule //end of module prog
```

No.

Date

Coursework

Introduction

This exercise is done individually and the assessment is:
- By formal report describing the final design, its development, implementation and testing.
- By a laboratory demonstration of the final design on an Altera FPGA development system.

Specification

The objective of this exercise is to design an n-bit implementation of picoMIPS

The design should be as small as possible in terms of FPGA resources but sufficient to implement the affine transformation algorithm described blow.

The size cost function of the design is defined as follows:

cost = number of Logic Elements used + max(0, number of embedded multipliers used − 2) + 30 × Kbits of RAM used)

Each logic Element has a flip-flop hence flip-flops are included in the above cost figure. The cost figure should be calculated for Altera Cyclone IV EP4CE115 and should be as low as possible. Altera Cyclone IV has 266 18×18 bit embedded hardware multipliers. If embedded multipliers are used in your implementation, up to two of them are 'free', i.e. they do not contribute to the size cost.

To demonstrate the cost figure of your design show in your report Altera Quartus synthesis statistics for Cyclone IV EP4CE115

Affine Transformation Algorithm 仿射变换

An affine transformation is a geometrical transformation that preserves co-linearity, i.e. all points lying on a line will also line after the transformation and distance ratios are preserved. For two-dimensional images, the general affine transformation can be expressed as:

$$\begin{vmatrix} x_2 \\ y_2 \end{vmatrix} = A \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} + B$$

where [x1, y1] are the coordinates of a pixel before the transformation and [x2, y2] – after the transformation.

The 2x2 matrix A and the two-element vector B provide the transformation coefficients. For example, a pure translation of pixels occurs if:

$A = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$, $B = \begin{vmatrix} b_1 \\ b_2 \end{vmatrix}$ Similar, the following coefficients implement pure scaling: $A = \begin{vmatrix} a_{11} & 0 \\ 0 & a_{22} \end{vmatrix}$, $B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$

In practice different affine transformations are combined to produce a complex transformation.

Implementation of affine transformation in picoMIPS

You are required to develop both a smallest possible picoMIPS architecture and a machine-level program for the general affine transformation. The 6 constants that define the transformation must be included as immediate literals in your program. You can chose one out of two sets of transformation constants as outlined below in Section "Data Sets". Pixel coordinates must be read from the switches SW0–SW7 on the FPGA development system and the resulting pixel coordinates after the transformation displayed on the LEDs LED0–LED7. Switch SW8 provides handshaking functionality as described in the pseudocode below. Switch SW9 should act as an active low RST

Pseudocode

1. Wait for coordinate x1 by polling switch SW8. Wait while SW8 = 0. When SW8 becomes 1 (SW8 = 1) read the coordinate x1 from SW0–SW7
2. Wait for switch SW8 to become 0
3. Wait for coordinate y1 as specified in step 1
4. Wait for SW8 to become 0
5. Execute the affine transformation algorithm and display coordinate x2 on LED0–LED7
6. Wait until SW8 becomes 1
7. Display coordinate y2 on LED0–LED7
8. Wait until SW8 becomes 0
9. Go to step 1

No.

Date

Input/Output

The input/output functionality can be implemented in several different ways. For example, you can design your own IN and OUT instructions for reading/writing data using external ports. To use fewer hardware resources, you can consider connecting the ALU output, or a register output directly to the LEDs. You could consider dedicating a specific register number, e.g. register 1 to the input port. In this way, an ADD instruction can be used to read data, e.g. ADD %5, %0, %1 would store the input data in register %5.

Data sets

In your implementation, choose one of the two following data sets:

$A = \begin{bmatrix} 0.25 & 0.5 \\ -0.5 & 0.75 \end{bmatrix}$ $B = \begin{bmatrix} 20 \\ -20 \end{bmatrix}$ or $A = \begin{bmatrix} 0.5 & -0.875 \\ -0.875 & 0.75 \end{bmatrix}$ $B = \begin{bmatrix} 5 \\ 12 \end{bmatrix}$

Suggested data formats Affline transformation and fixed-point representation.

For the two-dimensional affline transformation implemented in your picoMIPS: $\begin{vmatrix} x_2 \\ y_2 \end{vmatrix} = A \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} + B$

Use the following data formats. Pixel coordinates [x1,y1] and [x2,y2] and coefficients of vector B are 8-bit 2's complement signed integers, i.e. their values are in the range -128..+127.

The coefficients of matrix A are 2's complement signed fixed-point fractions in the range -1...+1 - 2^(-7) i.e. they are 2's complement fractional numbers with the radix point positioned after the most significant bits. Therefore the weights of the individual bit are:

Therefore the weights of the individual bits are:

	-2^0	2^(-1)	2^(-2)	2^(-3)	2^(-4)	2^(-5)	2^(-6)	2^(-7)
Bit position	7	6	5	4	3	2	1	0

When coefficients of the matrix A are multiplied by pixel coordinates, a double-length 16-bit product is obtained which is a 2's complement number with the radix point positioned after the 9-th bit.

Note however that the result [x2,y2] must be an 8-bit 2's complement whole number

Binary multiplication examples

Binary multiplication of 2's complement 8-bit numbers yields a 16-bit result. As one of the number is represented in the range -1..+1-2^-7 and the other in the range -128...127, it is important to determine correctly which 8-bits of the 16-bit result represent the integer part which should be used for further calculations. The following examples illustrate which result bits represent the integer part.

[Example 1: 0.75 x 6]

-2^8 : 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 : 2^{-1} 2^{-2} 2^{-3} 2^{-4} 2^{-5} 2^{-6} 2^{-7}

Result (Integer

⇓

The new leading bit must now change from 2^7 to -2^7

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

```
while (sw8==0);
Load sw0-sw7 for x1
while (sw8==1);
Load sw0-sw8 for y1
while (sw8==0);
```

No.

Date

ALU Synthesis

RAM memory load and store instructions (the only instructions using the RAM (data memory) are load (LD) and store (ST))

ALU is used for address calculation
- typically ALU is programmed to copy operand B to Result : address is in register Rs or immediate

Data move from RAM to Rd (load) or from RA to RAM (store)

{ Load operation : Rd <= RAM[address]
{ store operation : RAM[address] <= Rd

Immediate Jump, absolute or relative If there is subroutine (know where to return)

Absolute jump : PC <= branch address, where branch address is : Instruction[p-1:0]

Relative jump : PC <= PC + Instruction[p-1:0], relative offset needs an adder within the PC

Indirect Jump, absolute or relative

Brun address is in a register 結構同上

ALS

a+b b

picoMIPS - Variable - Architecture Motif-specific Processor

Problems with standard many-core approaches: 1-processor size

- 100 ARM or Intel cores would take up far too much space to physically fit on a SoC that powers a mobile or tablet device.
- New architecture designs are required using smaller and simplier cores.
- Energy, Area and Performance Overhead of the Network-on-chip
  - Networking and shared resources in many-core
  - Huge number of 'wires' between cores and themselves
  - Complex switching - large space and large proportion of energy

2-Software

Current parallel programming techniques are not parallel enough
  - Current iPhone contains a dual-core processor and yet outperforms smartphones with much higher core counts.

Most programs are serial by nature
  - Standard architectures with 100s of cores cannot be used effectively

Backward compatibility of software
  - Smooth transitions from existing software to many-core implementations are difficult

Key design challenge in embedded mobile systems

Reduce energy consumption while delivering high performance

Promissing approach: application-driven variable-architecture processors
A methodology is presented to create an energy-efficient variable architecture processor that is tailored to application requirements
Case studies: Affine transformation of pixels
JPEG decompressor Chardware size is reduced four times compared with equivalent dedicated hardware.

Variable-architecture picoMIPS

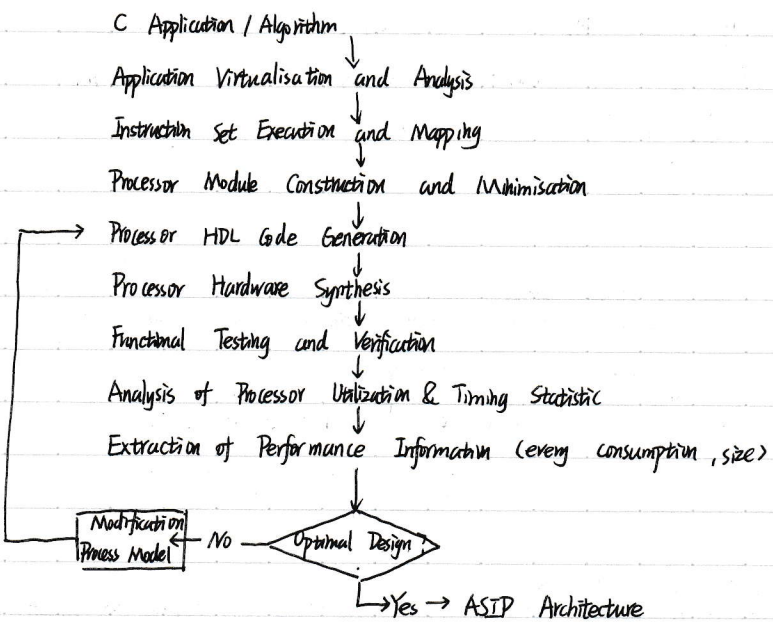
Embedded processors usually run specific, fixed application types (motifs 图案)
Proposed architecture for such applications: picoMIPS
picoMIPS:
- Inspired by MIPS, but not a MIPS clone or subset
- Variable size data width
- Variable structure and variable size hardware blocks:
  - ALU, register file, program memory, instruction decoder, I/O
- Application specific instruction format and instruction set
- Only essential data storage, instructions and ALU functions are implemented during synthesis.
- More versatile than typical FPGA soft cores, e.g. Altera NIOS

picoMIPS - can be customised as a motif-oriented processor

User specifies the width of the data bus n, typically 8, 16, 32, 64 bits and the program counter size p.

User specifies the instruction size i, instruction set format and instruction set to be implemented in the decoder.

Processor synthesis flow



PicoMIPS - Summary

A variable-architecture application-specific processor design methodology is proposed
- Oriented towards extremely low energy consumption
- Can be implemented in a many-core small worker environment

Suitable for embedded applications

Potential for extreme energy savings in ASIC implementations
- Standard energy saving techniques can be used, such as near-threshold design, dynamic voltage and frequency scaling etc.

Further work
- Software toolkit to automate the instruction set and architecture generation as well as translation of high-level code to machine-level instructions.

No.

Date

Embedded Hardware Multipliers

Unsigned binary multiplication

$$
\begin{array}{rrrrrrrr}
 & & & A_3B_3 & A_2 & A_1 & A_0 \\
 & & & \times B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
\end{array}
$$

2N-bit product

[Example] 7 × 5 unsigned

```
  111   M  multiplicand
 X101   Q  multiplier
  111      (ADD 7, Q0=1)
  000      (Don't ADD, Q1=0)
 111       (ADD 7 shifted by 2 bits, Q2=1)
100011     (result=35)
```

Hardware implementations

① Sequential – adder, shift-register, state machine

② Combinational – adders and AND gates for partial product AB_i

Same example cycle by cycle for sequential hardware implementation

M = 1 1 1 (N=7)

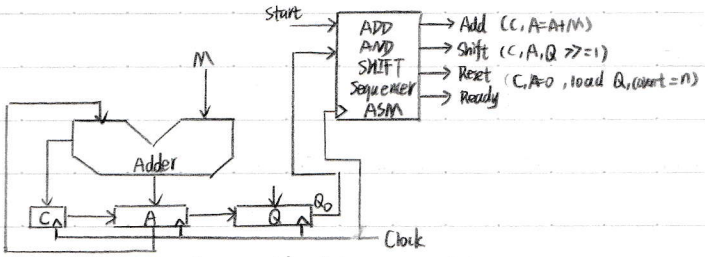
C	A	Q=5 (current operand)	
0	0 0 0	1 0 1	Initial Values
0	1 1 1	1 0 1	ADD : A := A+M
0	0 1 1	1 1 0	SHIFT (cycle 1)
0	0 0 1	1 1 1	SHIFT (cycle 2)
1	0 0 0	1 1 1	ADD : A := A+M
0	1 0 0	0 1 1	SHIFT (cycle 3)
A,Q =	0 1 0 0	0 1 1	= 35

001 + 111 → 1000 overflow

共有物路

Note: Here the double length accumulator AQ shares storage with multiplier Q

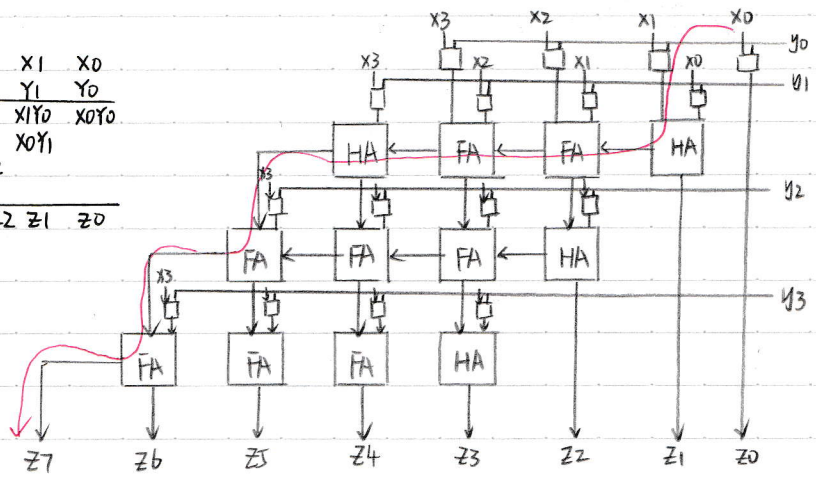
Sequential unsigned multiplier - hardware implementation



C, A, Q can righ shift by one bit.

Combinational unsigned multiplier

$$
\begin{array}{rrrrrrrr}
 & & & X_3 & X_2 & X_1 & X_0 \\
 & & & \times Y_3 & Y_2 & Y_1 & Y_0 \\
\hline
 & & & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\
 & & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 & \\
 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 & & \\
X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 & & & \\
\hline
Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0
\end{array}
$$



Propagation Delay ~2N adders

Signed Multiplication

Can extend addition to 2n bits

负X正 → n cycles

Negative multiplicand, positive multiplier – n cycles

-5	1	1	1	1	1	0	1	1	X
5	0	0	0	0	0	1	0	1	=
	1	1	1	1	1	0	1	1	+
	0	0	0	0	0	0	0		+
	1	1	1	0	1	1			+
	0	0	0	0	0				+
-25	1	1	0	0	0	1	1	1	=
	-128	64	32			4	2	1	

Positive or negative multiplicand, negative multiplier

– 2n cycles, but can be reduced to n cycles using a special treament of the upper half

7	0	0	0	0	0	1	1	1	X
-3	1	1	1	1	1	1	0	1	=
	0	0	0	0	0	1	1	1	+
	0	0	0	0	0	0	0	0	+
	0	0	0	1	1	1	0	0	+
	0	0	1	1	1	0	0	0	+
	0	1	1	1	0	0	0	0	+
	1	1	1	0	0	0	0	0	+
	1	1	0	0	0	0	0	0	+
	1	0	0	0	0	0	0	0	+ ← 反码
-21	1	1	1	0	1	0	1	1	=
	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	

Combinational signed multiplication extending addition to 2n bits – much exact circuitry

				X3	X2	X1	X0
				Y3	Y2	Y1	Y0
	X3Y0	X3Y0	X3Y0	X3Y0	X3Y0	X2Y0	X1Y0	X0Y0
+	X3Y1	X3Y1	X3Y1	X3Y1	X2Y1	X1Y1	X0Y1	
+	X3Y2	X3Y2	X3Y2	X2Y2	X1Y2	X0Y2		
−	X3Y3	X3Y3	X2Y3	X1Y3	X0Y3			
	Z7	Z6	Z5	Z4	Z3	Z2	Z1	Z0

0111 7

1101 -3

0111

0000

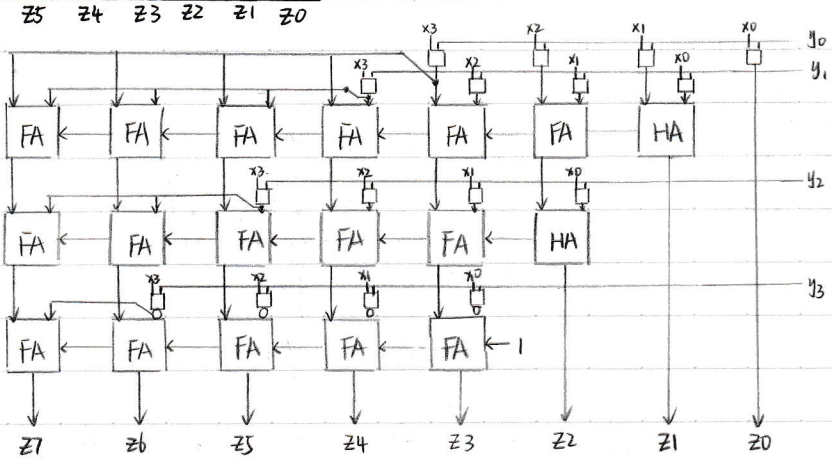
1001

11101011

6 5 4 3 2 1 0

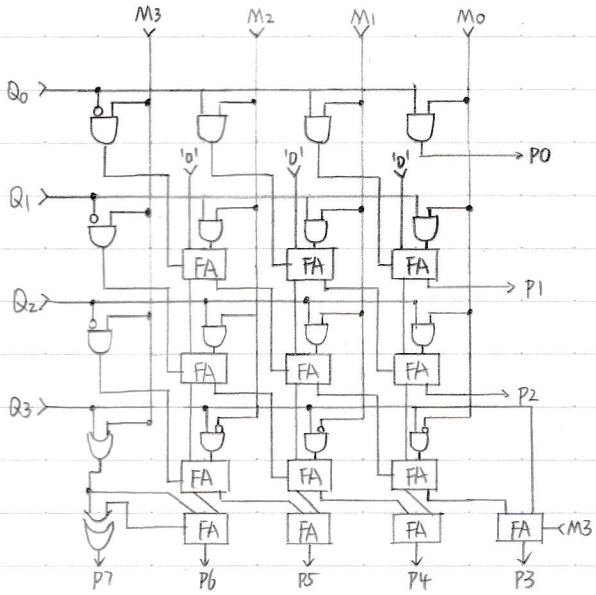
1 1 0 0 0 1 1

$-2^6+2^5+2^1+2^0 =$



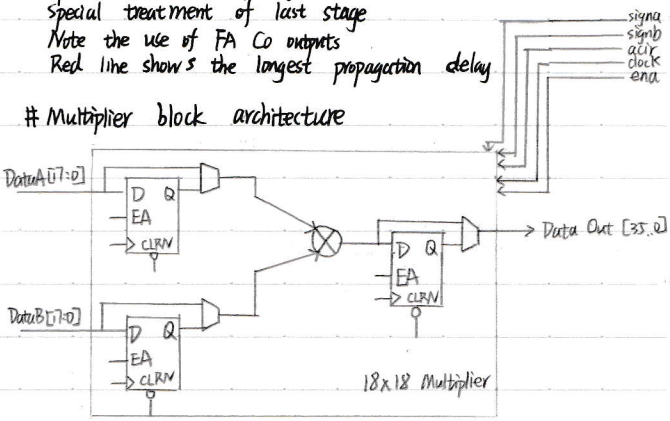
有符号单比器最后一位为减法

Hardware efficient combinational signed multiplier Baugh-Wooley multiplier

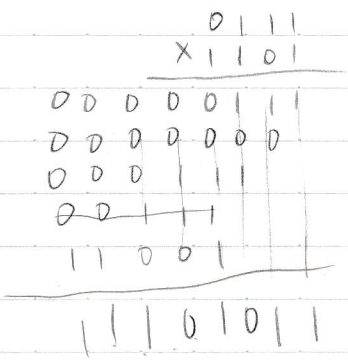


n-1 bit addition
Special treatment of sign bits
Special treatment of last stage
Note the use of FA Co outputs
Red line shows the longest propagation delay

Multiplier block architecture



Inferring multipliers from SystemVerilog code



Pipelined CPUs

Key features of RISC and pipelines

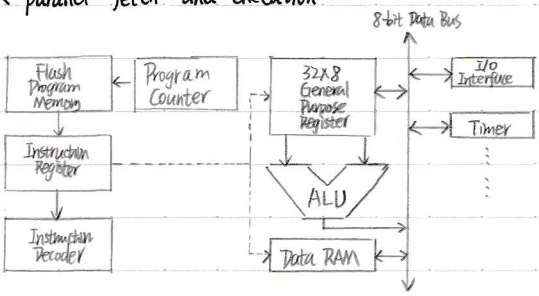
- Small and simple instruction set.
- Most instructions execute in 1 single clock cycle (no microcode)
- Memory access instructions only transfer data: memory <-> registers
- ALU operations are register to register
- Large number of general purpose registers

- RISC lends itself easily to efficient instruction execution pipeline
  - Pipelines accelerate performance
- Pipelines are not always used in embedded architectures
  - Hardware overheads are necessary to implement and control a pipeline.
  - But many embedded ~~hardware~~ processors are pipelined, e.g. ARM Cortex cores.
    - ARM Cortex M3 - 3 stage pipeline
    - ARM Cortex A9 - dynamic length, 9-11 stage pipeline.

Even simplest microcontrollers use pipelines

- AVR has a two-stage pipeline: parallel instruction fetches and instruction executions
- This is enabled by the Harvard architecture of the AVR and the use of instruction register.
- This basic pipelining results in fast execution, single-clock cycle per instruction.

AVR parallel fetch and execution



Instruction loaded into Instruction Register executes

At the same time next instruction is fetched from Program Memory

Pipeline hazards

Instruction cycle broken into n phases (one execution stage per phase)
- eg, Fetch, Decoder, ReadOPs, Execute1, Execute2, WriteBack.

A new instruction is fetched each phase
Maximum speed gain is n times
Pipeline hazards reduce the ability to achieve a gain of n times.

[Type of hazards]

- Resource
  - Hazard occurs when instruction needs a resource being used by another instruction.
- Data
  - RAW (read-after-write hazard: read is requested before write has finished)
  - WAR (write-after-read hazard: write is requested before read is finished)
  - WAW (write-after-write hazard: writes occur in an unintended order)
- Control
  - Hazard occurs when a wrong fetch decision at a branch results in an extra instruction fetch and a pipeline flush.

No.

Date

RAW: 一条指令的操作数来自之前指令的结果
WAR: 一条指令要将结果写入某个寄存器中，但这个寄存器正在被其它指令读取
WAW: 如果两条指令都要将结果写到同一个寄存器中，那么后面的指令必须等到前面的指令写完。

Data hazards

Dependences between instructions may cause data hazards when Instr1 and Instr2 are so close that their overlapping within the pipeline would change their access order to Reg.

Three types of data hazards:

Read After Write (RAW): Instr2 tries to read operand before Instr1 writes it.
Write After Read (WAR): Instr2 tries to write operand before Instr1 reads it.
Write After Write (WAW): Instr2 tries to write operand before Instr1 writes it.

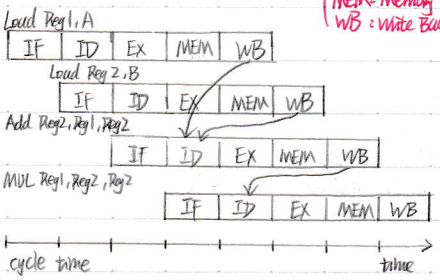
R1 = R2 + R3
R5 = R1 + R4 — true dependence

R1 = R2 + R3
R2 = R5 + R4 — anti dependence

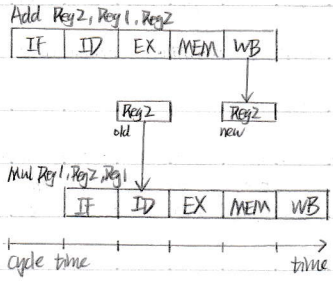
R1 = R2 + R3
R1 = R5 + R4 — output dependence

Data access hazard - example

IF : Instruction Fetch
ID : Instruction Decode
EX : Execute
MEM: Memory Access
WB : Write Back



Pipeline conflict due to data dependency hazard

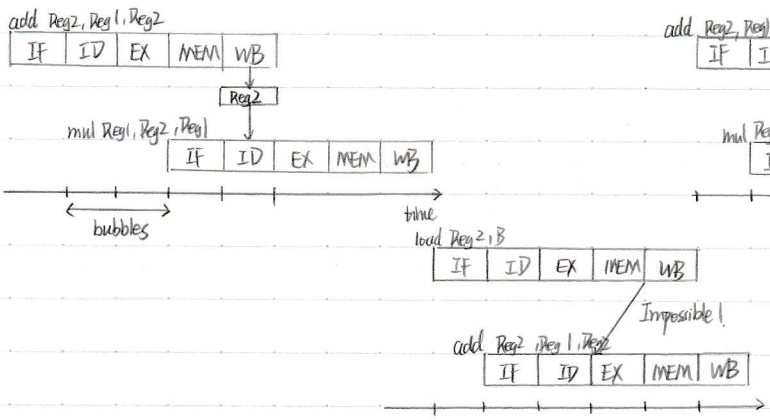


Solutions to data access and data dependancy hazards

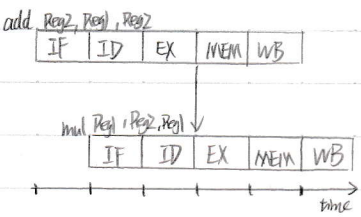
Software solutions (Compiler Scheduling):
- Putting no-op instructions after each instruction that may cause a hazard
- Instruction scheduling: rearrange code to reduce no-ops

Hardware solutions
- Hazard detection hardware is necessary.
- Interclocking: stall pipeline for one or more cycles
- Forwarding: two types of forwarding:
  - The ALU result of Instr1 in EX stage can immediately be forwarded back to ALU input of EX stage as an operand for Instr2
  - The memory load data from MEM stage can be forwarded to ALU input of EX stage.
- Forwarding with interclocking 暂停
  - Assume that Instr2 is data dependent on the load instruction Instr1 then Instr2 has to be stalled until the data loaded by Instr1 becomes available.

Hardware solution to data dependency hazard - interclocking



Solution to data dependency hazard - forwarding

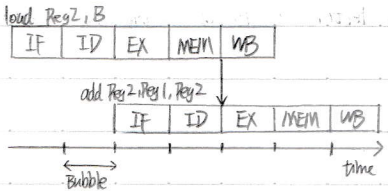


Example of data dependency hazard not resolvable by forwarding

直接将结果输入EX继续执行

用 bubble 延迟指令

Example of data dependency hazard - resolved by forwarding and stalling



Control hazards

Control hazards occur when a wrong ~~function~~ fetch decision results in a new instruction fetch and the pipline being flushed
Dealing with control hazards may require significant hardware overheads.
Solution include:

- Multiple Pipeline streams
- Prefetching the branch target
- Using a loop buffer
- Branch Prediction
- Delayed Branch
- Reordering of Instructions
- Multiple copies of Registers (backups)

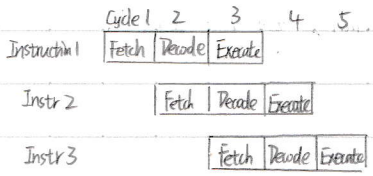
3-stage ARM Cortex M3 pipeline

Each instruction is executed in three stages

① Fetch: instruction is fetched from memory and replaced in pipeline
② Decode: instruction is decoded and data-paths signals prepared for next cycle.
③ Execute: instruction reads ALU operands from registers and writes ~~ALU~~ result to destination register.

ARM pipeline is linear: processor throughput is one instruction per clock cycle while individual instruction takes three clock cycles.

- When a branch instruction is fetched, a pipeline faces difficulties, because wrong instructions may have been fetched into the pipeline. If this occurs, pipeline flushes and has to be refilled. ARM pipeline maintains high performance and predicts branch behaviour because PC address is calculated 2 instructions ahead of current instruction.



No.

Date

MIPS Pipeline

Pipelined version of MIPS is a good example of a pipelined RISC architecture (relative simple and real)

[RISC] MIPS Instructions

Load/Store Instructions

LB	Load Byte	(I-type)
LBU	Load Byte Unsigned	
LH	Load Halfword	
LHU	Load Halfword Unsigned	
LW	Load Word	
LWL	Load Word Left	
LWR	Load Word Right	
SB	Store Byte	
SH	Store Halfword	
SW	Store Word	
SWL	Store Word Left	
SWR	Store word Right	

Shift Instructions

SLL	Shift Left Logic
SRL	Shift Right Logic
SRA	Shift Right Arithmetic
SLLV	Shift Left Logic Variable
SRLV	Shift Right Logic Variable
SRAV	Shift Right Arithmetic Variable

Multiply/Divide Instructions

MULT	Multiply (FP type)
MULTU	Multiply Unsigned
DIV	Divide
DIVU	Divide Unsigned
MFHI	Move From HI
MTHI	Move to HI
MFLO	Move From LO
MTLO	Move To LO

Coprocessor Instructions

LWCz	Load Word to Coprocessor
SWCz	Store Word to Coprocessor
MTCz	Move To Coprocessor
MFCz	Move From Coprocessor
CTCz	Move Control to Coprocessor
CFCz	Move Control from Coprocessor
COPz	Coprocessor Operation
BCzT	Branch on Coprocessor z True
BCzF	Branch on Coprocessor z False

Arithmetic Instructions (ALU immediate)

ADDI	Add Immediate
ADDIU	Add Immediate Unsigned
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
ANDI	AND Immediate
ORI	OR Immediate
XORI	Exclusive-OR Immediate
LUI	Load Upper Immediate

Arithmetic Instructions (3-operand, R-type)

ADD	Add
ADDU	Add Unsigned
SUB	Subtract
SUBU	Subtract Unsigned
SLT	Set on Less Than
SLTU	Set on Less Than Unsigned
AND	AND
OR	OR
XOR	Exclusive-OR
NOR	NOR

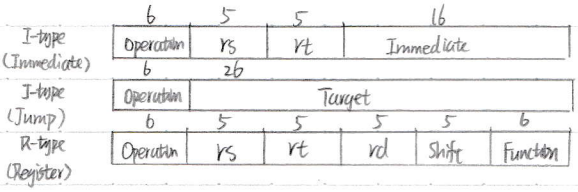
Jump and Branch Instructions

J	Jump
JAL	Jump and Link
JR	Jump to Register
JALR	Jump and Link Register
BEQ	Branch on Equal
BNE	Branch on not Equal
BLEZ	Branch on Less than or equal to Zero
BGTZ	Branch on Greater than Zero
BLTZ	Branch on Less than Zero
BGEZ	Branch on Greater than or equal to Zero
BLTZAL	Branch on Less than Zero and Link
BGEZAL	Branch on Greater Than or Equal to Zero and Link.

Special Instructions

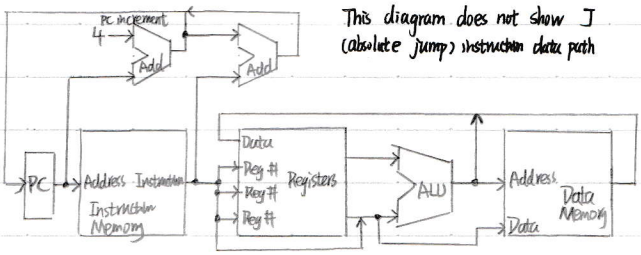
SYSCALL	SYSTEM CALL
BREAK	Break

MIPS Instruction Formats

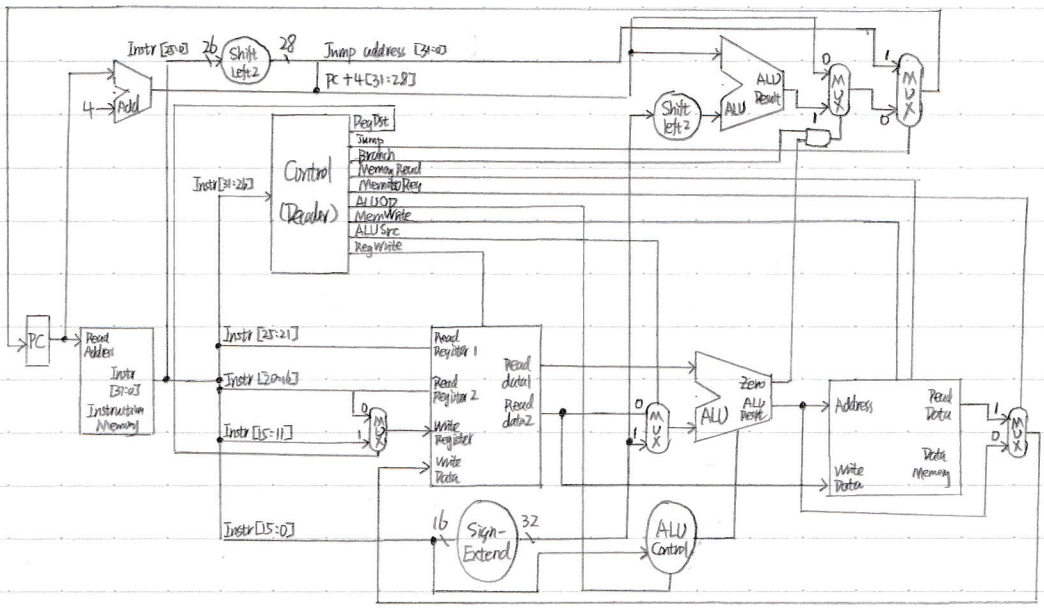


Operation	Operation code
rs	Source register specifier
rt	Source / destination register specifier
Immediate	Immediate, branch, or address displacement
Target	Jump target address
rd	Destination register specifier
Shift	Shift amount
Function	ALU / shift function specifier

Single-cycle MIPS Architecture



完全体



No.

Date

最先进的

RISC-V A State-of-the-art open-source RISC with potential for embedded applications

RISC-V features

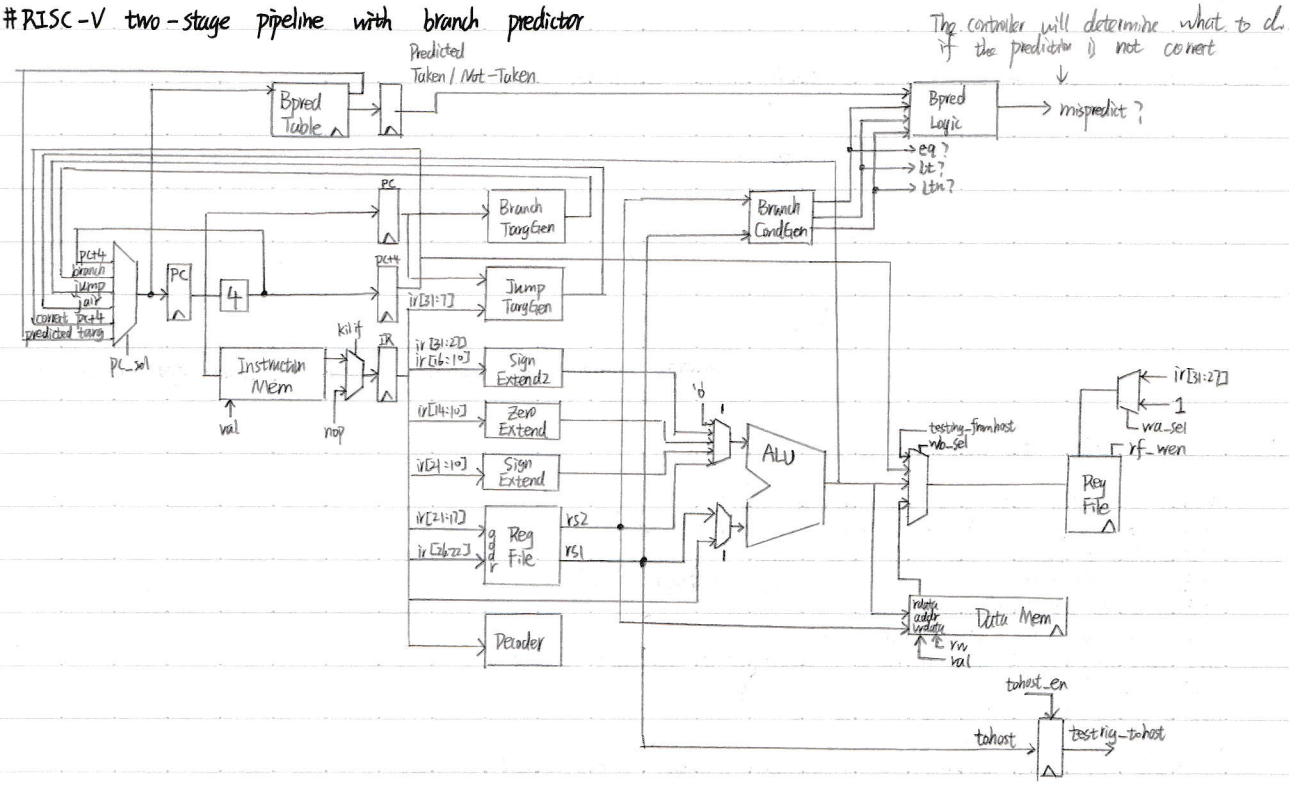
Small CPU
- many have 16 registers
- basic version includes only 33 instructions

Ideas borrowed from SPARC
- register 0 is a constant 0
- presence of instruction register, gives rise to a simple, two-stage fetch-execute pipeline

Ideas borrowed from MIPS
- No status register
- Conditional branches evaluate branch condition in the same cycle
- Memory-mapped I/O

Novel ideals
- ALU intensionally lacks condition codes, branch conditions are evaluated by separate logic
  - But the lack of carry bit complicates multiple precision arithmetic
  - RISC V does not detect or flag most arithmetic errors
- Extra space is reversed for new instructions, future extensions but also user defined instructions

RISC-V two-stage pipeline with branch predictor



Two Reg File modules above are the same modular

KOKUYO

No.

Date

Need more

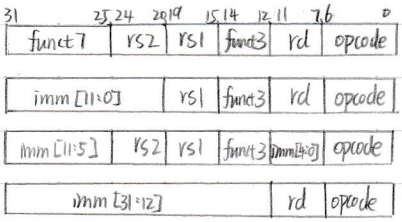
RISC-V branch prediction logic

The predictor uses a four entry table Bpred. Each entry is a <pc+4, target> pair.
In the fetch stage, the predictor uses PC+4 to index the table, and if the PC's match then this is a hint hit
If the PC's do not match, then this is a miss and PC+4 is used as the next PC instead of target.
The hint hit/miss signal is pipelined to the execute stage. This signal tells the control logic if the branch was predicted taken or not taken. Actions of the control logic are:

Predicted	Actual	Mispredict?	Action to take
taken	taken	no	no operation
not-taken	taken	yes	kill instr in fetch, update table, pc := branch or jump tag
taken	not-taken	yes	kill instr in fetch, do not update table, pc := correct PC+4
not-taken	not-taken	no	no action required.

If the branch was predicted not-taken, and it was actually taken (i.e. a loop was entered), then table Bpred is updated by adding the appropriate PC+4 and branch or jump target.
If the branch was predicted taken ~~or not taken~~ and it was actually not-taken (i.e. a loop exits), then there is no need to update the table.

Instruction format (RV32I - RISC-V 32-bit Integer Instruction Set)



R-type ← rd ← rs1 op rs2 (register only instructions)

I-type ← rd ← rs1 op imm (register and immediate)

S-type ← memory store instructions; Loads are implemented as I-type

U-type ← instructions involving PC, e.g. JAL (jump and link) and other instructions involving large immediates, e.g. LUI (load upper immediate)

Comments regarding the instruction set.

Extend instruction sets are also defined, e.g.:
- RV64I, RV128I (64-bit and 128-bit integer instructions)
- Extensions for integer multiplication and division.
- Extensions for floating point (single, double and quadruple precision
- Extensions for bit manipulation.
- Extensions for vector architectures (SIMD)

xi, where i = 0..32, is used in RISC-V assembler to represent register i.
Register x0 is always 0
NOP is implemented as: ADDI x0, x0, 0
J is implemented as: JAL x0, address
- JAL rd, address normally calls a subroutine, where PC+4 is saved in rd.

stand alone, If you want high-performance → hardcore
(Not always true)

No.
Date

Soft processor cores in embedded systems

Microprocessors in embedded systems

Microprocessors are everywhere
A modern car has around 40 to 100 microprocessors
- Engine control, break systems, active suspension (Need powerful, 32-bit processors)
- Even motorised (使机械化) window control has a microprocessor

Designer face many challenges
- Which parts of a system on chip should be implemented as programmable, combinational and sequential logic
- Which parts should be implemented as programs on a microprocessor core?
- What processor architecture should be used?
  - Little point in using a 32-bit MIPS with double precision floating point if one is designing a burglar alarm (防窃)

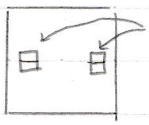
Processor cores

Can be classified as 'hard' or 'soft'
- 'hard' core is an embedded processor, e.g. ARM, sourrounded by FPGA's memory and programmable logic
- 'soft' core is synthesized from HDL code using standard FPGA's memory and logic

The picoMIPS we are designing is a soft core
Hard cores are less configurable but have higher performance than soft cores

Hard processor cores

Example of hard cores on FPGAs:
- Altera offers ARM core in some of their FPGAs, e.g. Cyclone V
- Xilinx also offers ARM or PowerPC



Two PowerPC hard cores on a Xilinx Virtex II die
两个硬核，不是从代码编译而来而是直接就在板子上

Soft cores

Use existing FPGA logic elements
Flexible, feature rich, reconfigurable
Customized memory size, data path width, ALU functionality, number and types of peripherals
Typically have slower ~~on~~ clock rates
Are said to consume more power than their hard-core counterparts (Not always true)
[Examples]
- Altera: NIOS II - one of the most popular
- Xilinx: MicroBlaze, PicoBlaze
- Lattice: LatticeMico8
- ARM Cortex M1 - a third party core offered to FPGA vendors 供应商

There are many open-source soft cores, e.g.
- SPARC - GPL license (www.simplyrisc.com)
- OpenRISC - also GPL (e.g. opencores.org)
- DSPuva16

Development Tools

A dedicated CAD tool, specific to the particular core, is used to specify processors parameters.
- Register file size, hardware arithmetic, floating point, interrupts, I/O hardware, custom functions
- User programs, typically written in C or C++, are compiled by a custom compiler provided by the CAD tool and included in the synthesis

The tool outputs a synthesizable HDL model
- Normally the HDL code is protected and not viewable.

After any user-specified hardware is added, the design is synthesized using a standard synthesis tool.
- Altera Quartus integrates specification for the NIOS processor into a single tool used for general purpose synthesis
- Altera also provides a SOPC (System on Programmable Chip) Builder for advanced embedded system design.

No.

Date

Altera NIOS II Soft Core Processor

A general purpose RISC with a Harvard memory architecture

Features a 32-bit Instruction Set Architecture (ISA), 32-general-purpose registers, single-instruction 32x32 multiply and divide operations, and dedicated instructions for 64-bit and 128-bit products of multiplication.

Nios II has a performance of more than 150 的 Dhrystone MIPS (DMIPS) 测试程序 on Stratix family of FPGAs

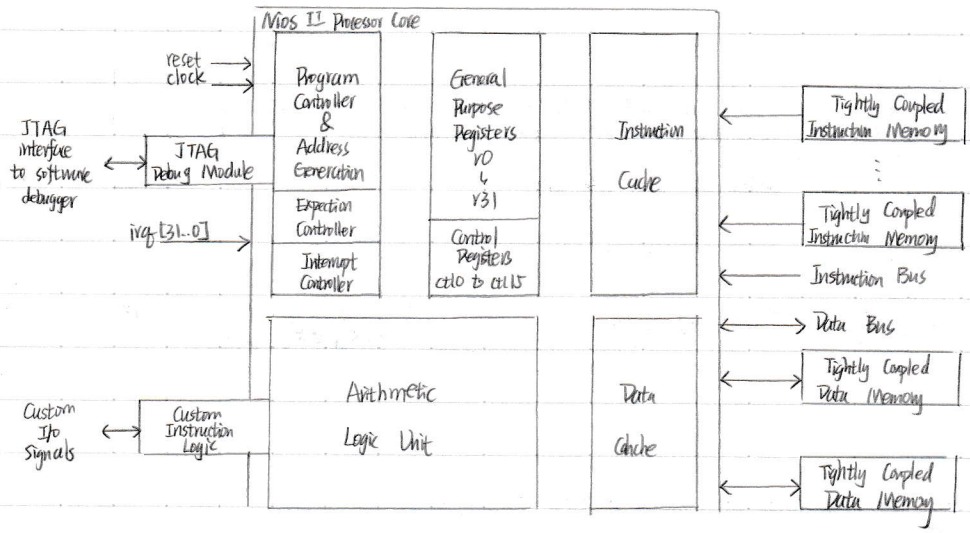
Comes in three versions: economy, standard and fast core.

Each core version has a different number of pipeline stages, instruction and data the cache memory and hardware components for X and

Each core implementation varies in size and performance depending on the features that are selected.

Adding peripherals with the Nios II Processors is done through the Avalon Interface Bus which contains the necessary logic to interface

processor with off-the-shelf (现成的) IP cores or custom designed peripherals.



No.

Date

MIPS Instruction Reference

This is a description of the MIPS instruction set, their meanings, syntax, semantics, and bit encodings. The syntax given for each instruction refers to the assembly language syntax supported by the MIPS assembler. Hyphens ("-") in the encoding indicate "don't care" bits which are considered when an instruction is being encoded.

General purpose registers (GPRs) are indicated with a dollar sign ($). The word SWORD and UWORD refer to 32-bit signed and 32-bit unsigned data types.

The manner in which the processor executes an instruction and advances its program counters is as follows:

1. execute the instruction at PC
2. copy nPC to PC
3. add 4 or the branch offset to nPC

This behavior is indicated in the instruction specifications below. For brevity, the function advance_pc(int) is used in many of instruction descriptions.

```
void advance_pc (SWORD offset)
{
    PC = nPC;          ;Each address in memory stores one byte and each MIPS instruction requires four bytes of memory
    nPC += offset;
}
```

Note: All algorithms arithmetic immediate values are sign-extended. After that, they are handled as signed or unsigned 32 bit numbers, depending upon the instruction. The only difference between signed and unsigned instructions is that signed instructions can generate an overflow expection and unsigned instructions can not.

[Instruction Descriptions]

ADD - Add (with overflow)

Description: Adds two registers and stores the result in a register
Operation: $d = $s + $t ; advance_pc(4);
Syntax: add $d, $s, $t
Encoding: 0000 00ss ssst tttt dddd d000 0010 0000

ADDI - Add Immediate (with overflow)

Description: Adds a register and a sign-extended immediate value and stores the result in a register
Operation: $t = $s + imm ; advance_pc(4);
Syntax: addi $t, $s, imm
Encoding: 0010 00ss ssst tttt iiii iiii iiii iiii

ADDIU - Add immediate unsigned (no overflow)

Description: Adds a register and a sign-extended immediate value and stores the result in a register
Operation: $t = $s + imm ; advance_pc(4);
Syntax: addiu $t, $s, imm
Encoding: 0010 01ss ssst tttt iiii iiii iiii iiii

ADDU - Add unsigned (no overflow)

Description: Adds two registers and stores the result in a register
Operation: $d = $s + $t ; advance_pc(4);
Syntax: addu $d, $s, $t
Encoding: 0000 00ss ssst tttt dddd d000 0010 0001

AND - Bitwise and

Description: Bitwise ands two registers and stores the result in a register
Operation: $d = $s & $t ; advance_pc(4);
Syntax: and $d, $s, $t
Encoding: 0000 00ss ssst tttt dddd d000 0010 0100

No.

Date

ANDI - Bitwise and Immediate

Description: Bitwise ands a register and an immediate value and stores the result in a register
Operation: $t = $s & imm ; adv
Syntax: andi $t, $s, imm
Encoding: 0011 00ss ssst tttt iiii iiii iiii iiii

BEQ - Branch on equal

Description: Branch if the two registers are equal
Operation: if $s == $t advance_pc (offset << 2) ; else advance_pc (4) ;
Syntax: beq $s, $t, offset ← (pc jumps how many instructions)
Encoding: 0001 00ss ssst tttt iiii iiii iiii iiii

BGEZ - Branch on greater than or equal to zero

Description: Branches if the register is greater than or equal zero
Operation: if $s >= 0 advance_pc (offset << 2) ; else advance_pc (4) ;
Syntax: bgez $s, offset
Encoding: 0000 01ss ssso 0001 iiii iiii iiii iiii

BGEZAL - Branch on greater than or equal to zero link

Description: Branches if the gregister is greater than or equal to zero and saves the return address in $31
Operation: if $s >= 0 $31 = PC + 8 (or nPC + 4) ; advance_pc (offset << 2) ; else advance_pc (4) ;
Syntax: bgezal $s, offset
Encoding: 0000 01ss ssso 0001 iiii iiii iiii iiii

BGTZ - Branch on greater than zero

Description: Branches if the register is greater than zero
Operation: if $s > 0 advance_pc (offset << 2) ; else advance_pc (4) ;
Syntax: bgtz $s, offset
Encoding: 0001 11ss ssso 0000 iiii iiii iiii iiii

BLEZ - Branch on less than or equal to zero

Description: Branch if the register is less than or equal to zero
Operation: if $s <= 0 advance_pc (offset << 2) ; else advance_pc (4) ;
Syntax: blez $s, offset
Encoding: 0001 10ss ssso 0000 iiii iiii iiii iiii

BLTZ - Branch on less than zero

Description: Branches if the register is less than zero
Operation: if $s < 0 $31 = PC + 8 (or nPC + 4) ; advance_pc (offset << 2) ; else advance_pc (4) ;
Syntax: bltzal $s, offset
Encoding: 0000 01ss ssso 0000 iiii iiii iiii iiii

有无link：有类似于 call function
没有就是 go to

nPC = (PC & 0xf0000000) | (target << 2);

J有度，offset is limited！

pc+8 or (nPC+4)
↑ nPC在上次调用时已经+4，这再加4，等于在 PC=nPC时，PC+8 ~

BLTZAL - Branch on less than zero and link

Description: Branches if the register is less than zero and saves the return address in $31
Operation: if $s<0 $31 = PC+8 (or nPC+4); advance_pc (offset << 2); else advance_pc (4);
Syntax: bltzal $s, offset
Encoding: 0000 01ss ssss 0000 iiii iiii iiii iiii

BNE - Branch on not equal

Description: Branches if the two registers are not equal
Operation: if $s != $t, advance_pc (offset << 2); else advance_pc(4)
Syntax: bne $s, $t, offset
Encoding: 0001 01ss ssst tttt iiii iiii iiii iiii

DIV - Divide

Description: Divides $s by $t and stores the quotient (商) in $LO and the reminder in $HI
Operation: $LO = $s / $t; $HI = $s % $t; advance_pc(4);
Syntax: div $s, $t
Encoding: 0000 00ss ssst tttt 0000 0000 0001 1010

DIVU - Divide unsigned

Description: Divides $s by $t and stores the quotient in $LO and the reminder in $HI
Operation: $LO = $s/$t; $HI = $s % $t; advance_pc(4);
Syntax: divu $s, $t
Encoding: 0000 00ss ssst tttt 0000 0000 0001 1011

J - Jump

Description: Jumps to the calculated address
Operation: PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax: j target
Encoding: 0000 10ii iiii iiii iiii iiii iiii iiii

JAL - Jump and Link

Description: Jumps to the calculated address and stores the return address in $31
Operation: $31 = PC + 8 (or nPC + 4); PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax: jal target
Encoding: 0000 11ii iiii iiii iiii iiii iiii iiii

JR - Jump register

Description: Jumps to the address contained in register $s
Operation: PC = nPC; nPC = $s;
Syntax: jr $s
Encoding: 0000 00ss ssss 0000 0000 0000 0000 1000

No.

Date

LB - Load byte

Description: A byte is loaded into a register from the specified address
Operation: $t = MEM[$s + offset]; advance_pc(4);
Syntax: lb $t, offset($s)
Encoding: 1000 00ss ssst tttt iiii iiii iiii iiii

LUI - Load upper immediate

Description: The immediate values is shifted left 16 bits and stored in the register. The lower 16 bits are zeros.
Operation: $t = (imm << 16); advance_pc(4);
Syntax: lui $t, imm
Encoding: 0011 11-- ---t tttt iiii iiii iiii iiii

LW - Load word

Description: A word is loaded into a register from the specified address
Operation: $t = MEM[$s + offset]; advance_pc(4);
Syntax: lw $t, offset($s)
Encoding: 1000 11ss ssst tttt iiii iiii iiii iiii

MFHI - Move from HI

Description: The contents of register HI are moved to the specified register
Operation: $d = HI; advance_pc(4);
Syntax: mfhi $d
Encoding: 0000 0000 0000 0000 dddd d000 0001 0000

MFLO - Move from LO

Description: The contents of register LO are moved to the specified register
Operation: $d = LO; advance_pc(4);
Syntax: mflo $d
Encoding: 0000 0000 0000 0000 dddd d000 0001 0010

MULT - Multiply

Description: Multiplies $s by $t and stores the result in $LO
Operation: $LO = $s * $t; advance_pc(4);
Syntax: mult $s, $t
Encoding: 0000 00ss ssst tttt 0000 0000 0001 1000

MULTU - Multiply unsigned

Description: Multiplies $s by $t and stores the result in $LO
Operation: $LO = $s * $t; advance_pc(4);
Syntax: multu $s, $t
Encoding: 0000 00ss ssst tttt 0000 0000 0001 1001

No.

Date

NOOP - no operation

Description: Perform no operation
Operation: advance_pc(4)
Syntax: noop
Encoding: 0000 ... 0000

The encoding for a NOOP represents the instruction SLL $0, $0, 0 which has no side effects. In fact, nearly every instruction that has $0 as its destination register will have no side effect and can thus be considered a NOOP instruction.

OR - Bitwise or

Description: Bitwise logical ors two register's and stores the result in a register
Operation: $d = $s | $t ; advance_pc(4);
Syntax: or $d, $s, $t
Encoding: 0000 00ss ssst tttt dddd d000 0010 0101

ORI - Bitwise or immediate.

Description: Bitwise ors a register and an immediate value and stores the result in a register
Operation: $t = $s | imm ; advance_pc(4);
Syntax: ori $t, $s, imm
Encoding: 0011 01ss ssst tttt iiii iiii iiii iiii

SB - Store Byte

Description: The least significant byte of $t is stored at the specific address
Operation: MEM[$s+offset] = (0xff & $t); advance_pc(4);
Syntax: sb $t, offset($s)
Encoding: 1010 00ss ssst tttt iiii iiii iiii iiii

SLL - Shift left logical

Description: Shift a register value left by the shift amount listed in the instruction and places the result in a third register. zero are shift in.
Operation: $d = $t << h ; advance_pc(4);
Syntax: sll $d, $t, h
~~Syntax~~ Encoding: 0000 00ss ssst tttt dddd dhhh hh00 0000

SLLV - Shift left logical variable

Description: Shift a register value left by the value in a second register and places the result in a third register. Zeros are shifted in.
Operation: $d = $t << $s ; advance_pc(4);
Syntax: sllv $d, $t, $s
Encoding: 0000 00ss ssst tttt dddd d--- --00 0100

STL - Set on less than (signed)

Description: If $s is less than $t, $d is set to one. It gets zero otherwise
Operation: if $s < $t $d=1 ; advance_pc(4); else $d=0; advance_pc(4);
Syntax: slt $d, $s, $t
Encoding: 0000 00ss ssst tttt dddd d000 0010 1010

No.

Date

SLTI - Set on less than immediate (signed)

Description: If $s is less than immediate, $t is set to one. It gets zero otherwise
Operation: If $s < imm $t=1; advance_pc(4); else $t=0; advance_pc(4);
Syntax: slti $t, $s, imm
Encoding: 0010 ssss tttt iiii iiii iiii iiii

SLTIU - Set on less than immediate unsigned

Description: If $s is less than the unsigned immediate, $t is set to one. It gets zero otherwise
Operation: if $s < imm $t=1; advance_pc(4); else $t=0; advance_pc(4);
Syntax: sltiu $t, $s, imm
Encoding: 0010 1sss ssst tttt iiii iiii iiii iiii

SLTU - Set on less than unsigned

Description: If $s is less than $t, $d is set to one. It gets zero otherwise
Operation: if $s < $t $d=1; advance_pc(4); else $d=0; advance_pc(4);
Syntax: sltu $d, $s, $t
Encoding: 0000 00ss ssst tttt dddd d000 0010 1011

SRA - Shift right algorithm

Description: Shift a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in
Operation: $d = $t >> h; advance_pc(4);
Syntax: sra $d, $t, h
Encoding: 0000 00-- ---t tttt dddd dhhh hh00 0011

SRL - Shift right logical

Description: Shift a register value right by the shift amount (shamt) and places the value in the destination register. Zeros are shifted in.
Operation: $d = $t >> h; advance_pc(4);
Syntax: srl $d, $t, h
Encoding: 0000 00-- ---t tttt dddd dhhh hh 00 0010

SRLV - Shift right logical variable

Description: Shifts a register value right by the amount specified in $s and places the value in the destination register. Zeros are shifted in.
Operation: $d = $t >> $s; advance_pc(4);
Syntax: srlv $d, $t, $s
Encoding: 0000 00ss ssst tttt dddd d000 0000 0110

No.

Date

SUB - Subtract

Description: Subtract two registers and stores the result in a register
Operation: $d = $s - $t ; advance_pc(4);
Syntax: sub $d, $s, $t
Encoding: 0000 00ss ssst tttt dddd d000 0010 0010

SUBU - Subtract unsigned

Description: Subtract two registers and stores the result in a register
Operation: $d = $s - $t ; advance_pc(4);
Syntax: subu $d, $s, $t
Encoding: 0000 00ss ssst tttt dddd d000 0010 0011

SW - Store word

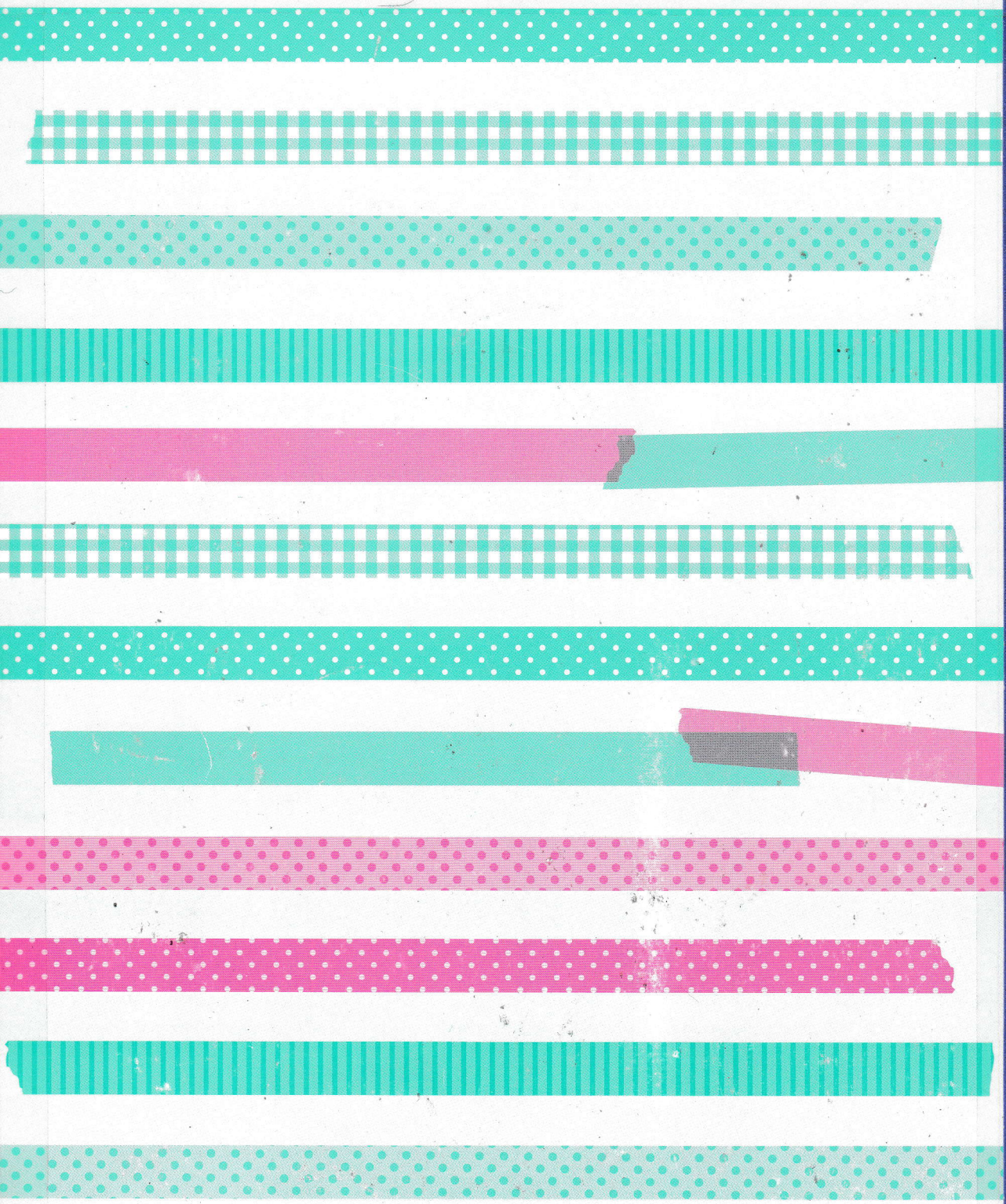
Description: The contents of $t is stored at the specific address.
Operation: MEM[$s + offset] = $t ; advan_pc(4);
Syntax: sw $t, offset($s)
Encoding: 1010 11ss ssst tttt iiii iiii iiii iiii

XOR - Bitwise exclusive or

Description: Exclusive ors two registers and stores the result in a register
Operation: $d = $s ^ $t ; advance_pc(4);
Syntax: xor $d, $s, $t
Encoding: 0000 00ss ssst tttt dddd d--- --10 0110

XORI - Bitwise exclusive or immediate

Description: Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation: $t = $s ^ imm ; advance_pc(4);
Syntax: xori $t, $s, imm
Encoding: 0011 10ss ssst tttt iiii iiii iiii iiii



Campus 无线装订本
B5 40页
WCN-CNB1430

6 937748 309932
MADE IN CHINA

国誉商业(上海)有限公司
http://www.kokuyo.cn/st/
上海市奉贤区人杰路128号
TEL：400-820-0798　FAX：021-3255-8508
产地：上海市　QB/T1438-2007合格
B5　252×179mm

● 采用日本进口纸张，纸质细滑，牢固不掉页。