I'll be proposing an architecture for a simple weather prediction computer called "Minified Computer Architecture (MCA)". It will be a 16-bit architecture. This will include complete processor, memory and cache architecture for a simple computer used for basic mathematic calculations for integers.

1.1 Processor architecture and instruction set:

1.1.1 Determine the processor components and the characteristics for each component.

MCA will manipulate binary data. Memory will be used to store program instructions and data associated with it in binary. There will be separate **Data memory** and **Instruction memory**. **Central Processing Unit (CPU)** will be used to fetch program instructions from instruction memory, decode each instruction that is fetched and execute the instruction as defined. CPU will consist of **registers**, arithmetic and logic unit (**ALU**) for performing various operations on data and a control unit (**CU**) to fetch and sequence operations and data. CPU will use a bus, which is set of wires, to communicate between memory.

1.1.2 Determine how many registers there are in your computer architecture.

General Purpose registers: R0 to R7 are 8 general purpose registers of size 8 bits.

Program counter: It's a 16 bit register to track the instruction under execution.

Accumulator registers: A0 to A1 are 2 accumulator registers used for compare operations

1.1.3 Determine the instructions that your proposed processor can handle.

Following are the 25 instructions that MCA can handle:

Ldd	Load a value into register
Ldr	Load a value from a register into register
Ldm	Load a value from memory into register
Sa	Store value in register into memory
Sah	Store higher order bits in register into memory
Sal	Store lower order bits in register into memory
Sr	Store value in register into a register

Srh	Store higher order bits in register into register
Srl	Store lower order bits in register into register
Cmpr	Compare contents of two registers
Cmpm	Compare contents of register with a value in memory
Addr	Add contents of two registers
Addm	Add contents of a register and a value in memory
Addd	Add contents of a register and a value
Subr	Subtract contents of two registers
Subd	Subtract contents of a register and a value in memory
Subm	Subtract contents of a register and a value
Mulr	Multiply contents of two registers
Mulm	Multiply contents of a register and a value in memory
Muld	Multiply contents of a register and a value
Divr	Divide contents of two registers
Divm	Divide contents of a register and a value in memory
Divd	Divide contents of a register and a value
stop	Stop instruction
noop	No operation instruction

1.1.4 Determine the instruction size

Instruction size is 32 bits.

- First 8 bits represent opcode. Since there are 25 instructions only 5 bits are required to represent opcodes but with 8 bits there will be enough room to add new instructions in future.
- Second 8 bits represent register. There are 10 (including accumulator) registers but only 3 bits are required to represent all registers but with 8 bits there will be enough room to add new registers.

- third 8 bits represent higher order byte if it is data/address or unused if the operation is on registers. Since it's 16-bit architecture 8 bits are required to represent higher order bytes in a memory address location.
- fourth 8 bits represent lower order byte if it is data/address or it used for a register. Since it's 16-bit architecture 8 bits are required to represent lower order bytes in a memory address location.

Instruction	Opcode	Register	Data/Address_H	Data/Address_L
ldd r1 512 # load data(512) into r1	0x00	0x01	0x00	0x200
ldr r2 r6 # load r6 into r2	0x01	0x02	NOT_USED	0x06
ldm r3 $0xABDF # load data from memory into r3	0x03	0x03	0xAB	0xDF
sa r4 $0xABDF # store contents of r4 to a address	0x10	0x04	0xAB	0xDF
sah r1 0xABDF # store higher order bits of r1 to an address	0x11	0x01	0xAB	0xDF
sal r1 0xABDF # store lower order bits of r1 to an address	0x12	0x01	0xAB	0xDF
sr r1 r2 # store contents of r1 to r2	0x13	0x01	NOT_USED	0x02
srh r1 r2 # store contents of r1 to r2 higher order bits	0x14	0x01	NOT_USED	0x02
srl r1 r2 # store contents of r1 to r2 lower order bits	0x15	0x01	NOT_USED	0x02
cmpr r1 r2 # compare contents of r1 with r2, sets 0(<), 1(=) and 2(>) in accumulator register	0x20	0x01	NOT_USED	0x02
cmpm r1 0xABDF # compare contents of r1 with data, sets 0(<), 1(=) and 2(>) in accumulator register	0x21	0x01	0xAB	0xDF
addr r1 r2 # add contents r1 and r2 and store in r1	0x30	0x01	NOT_USED	0x02
addm r1 $0xABDF # add contents r1 and memory and store in r1	0x31	0x01	0xAB	0xAB
addd r1 0xABDF # add data and r1 and store in r1	0x32	0x01	0xAB	0xDF
subr r1 r2 # sub contents r1 and r2 and store in r1	0x35	0x01	NOT_USED	0x02
subd r1 $0xABDF # sub contents r1 and memory and store in r1	0x36	0x01	NOT_USED	0x02
subd r1 0xABDF # sub data and r1 and store in r1	0x37	0x01	0xAB	0xDF
mulr r1 r2 # mul contents r1 and r2 and store in r1	0x40	0x01	NOT_USED	0x02
mulm r1 $0xABDF # mul contents r1 and memory and store in r1	0x41	0x01	NOT_USED	0x02
muld r1 0xABDF # mul data and r1 and store in r1	0x42	0x01	0xAB	0xDF

divr r1 r2 # div contents r1 and r2 and store in r1	0x45	0x01	NOT_USED	0x02
divm r1 $0xABDF # div contents r1 and memory and store in r1	0x46	0x01	NOT_USED	0x02
divd r1 0xABDF # div data and r1 and store in r1	0x47	0x01	0xAB	0xDF
stop	0xF3	0xFF	0xFF	0xFF

1.2 Instruction type and format:

1.2.1 Determine how many instructions types your processor will support.

MCA will support a total of 25 instructions. Among them 21 will be R-type instructions and 2 will be J-type instruction

1.2.2 Determine the format of each type.

Below are the instructions and its format. 1st 8 bits are for Opcode and 2nd 8 bits are for Register and if we are accessing memory location then 3rd and 4th 8 bits will correspond to higher and lower order bits of memory location. If it is register than 3rd 8 bits are not used and the 4th 8 bits represent a register.

We will be using a 32 bit fixed size instructions because they are easy to decode and pipeline. Variable size instruction would require complex instruction decoding implementation.

1.3 Processor architecture and characteristics

1.3.1 You need to determine single core vs multicore and the multithread support or not and why and how the multithread, that your proposed processor has, is supported.

MCA is supposed to run a single program at any time. That is the reason why there is no stack or stack pointer register. At any time, a single program is loaded into memory and the instructions are executed parallelly.

1.3.2 [TODO] Processor Pipelining stages (how many stages and the functions in each stage), pipelining hazards that may appear in your proposed processor, and how you are going to overcome these hazards.

High throughput is one of the requirements for MCA processors so a 5 stage pipeline will be used to increase the throughput. Since we operate on few registers we would have data hazards, structural hazards when multiple instructions trying to read from data memory and instruction memory and no control hazards because we

have jump instruction. We will be using 2 level local dynamic branch prediction to reduce stalls through control hazards. We will be using more memory ports and multiple ALU to eliminate structural hazard and use forwarding unit to eliminate data hazard.

2 Identify the memory structure for your proposed architecture.

2.1 Determine the memory structure used in your computer architecture.

As a weather prediction system, MCA should be able to run complex weather prediction algorithms to predict weather for next few days. MCA should be able to compute weather for a region at a faster rate. MCA should be able to efficiently read instructions from instruction memory and fetch operands from data memory. Since one of the requirements for MCA is high throughput, high memory bandwidth is required. Instructions and Data are stored in different memory. This means that CPU can simultaneously read and instruction for instruction memory and read or write data from or to the memory. Since Instruction width is 32 bit and data width is 16 bits it is better to have different memories for both for easier implementation. Instructions will be stored in read only memory and data will be stored in read-write memory.

2.2 Determine the size of your proposed memory.

As a weather prediction system, MCA need to analyze weather data over multiple days and run algorithms to predict weather for next few days. Since complex algorithms are used for weather prediction, MCA should be able to load and execute programs with huge number of instructions and store weather data for past few days, next few days and store intermediate computed data. This implies we need higher instruction and data memory. Since it's a 16 bit architecture we will be using 64Kb(2^16) instruction and data memory size.

2.3 Determine what instructions can access the memory and how.

Divm, Mulm, Ldm, Addm and Subm insturctions will access the memory using the data memory bus to fetch the data from memory.

2.4 Determine the cache levels.

As a weather prediction system, MCA need to execute complex algorithms. Complex algorithms will involve loops. In a loop we need to execute a single instruction multiple times we need to fetch that instruction multiple times from instruction memory. We will be using a 8Kb 2-way set L1 instruction cache to reduce instruction memory access time. MCA also need to fetch weather data for past few weeks, do some computation and store intermediate results into memory and fetch them again to finally predict the weather. We need a 16Kb 2-way set associative L1 data cache to reduce data memory access time.

2.5 Determine how cache works.

Whenever CPU fetch for next instruction, firstly, it checks in instruction L1 cache. If instruction is not present in L1 instruction cache then instruction need to be fetched from memory. After the instruction is fetched from memory we will be using LRU cache replacement strategy. Data cache works in the same way.

2.6 Propose a similar memory sharing protocol similar to the snoopy protocol.

MCA is a single processor computer. There is no cache coherence problem in single processor computers so Snoopy protocol is not applicable. But in general, other alternative to snoopy protocol is Directory-Based protocol where a common directory is used to maintain the coherence among the caches. Here the directory acts as a filter where the processors ask permission to load an entry form the primary memory to its cache memory. If an entry is changed the directory either updates it or invalidate the other caches with that entry.